Hardware Implementation of the Compression Function for Selected SHA-3 Candidates

A. H. Namin and M. A. Hasan

Department of Electrical and Computer Engineering
University of Waterloo
Waterloo, Ontario, Canada
Email: {anamin, ahasan}@uwaterloo.ca

Abstract. Hardware implementation of the main building block (compression function) for five different SHA-3 candidates is presented. The five candidates, namely Blue Midnight Wish, Luffa, Skein, Shabal, and Blake have been considered since they present faster software implementation results compared to the rest of the SHA-3 proposals. The compression functions realized in hardware create the message digest of size 256 bits. We report both ASIC and FPGA implementations. The results allow an easy comparison for hardware performance of the candidates.

Key words: Hash functions, SHA-3, Hardware, Blue Midnight Wish, Luffa, Skein, Shabal, Blake.

1 Introduction

Hash functions have many applications in cryptography mainly in digital signatures, message authentication codes (MACs), and other forms of authentication [1]. A public competition recently organized by the National Institute of Standard and Technology (NIST) is underway to select the new Standard Hash Algorithm (SHA-3) [2]. The competition is in response to a weakness found in the SHA family of hash functions [6]. Fifty one submitted algorithms have advanced to the first round of the competition. The source code of these algorithms in C language is available through the NIST SHA-3 webpage [2]. Source codes can be easily compiled for different platforms to measure the performance of each of the proposals. For a complete list of software performance one should refer to ECRYPT Benchmarking of All Submitted Hashes (EBASH) [4] or the SHA-3 zoo website [3].

Unfortunately small percentage of the proposals include hardware implementation results of their work. Also, for the limited number of proposals that were implemented in hardware, different technologies and platforms (ASIC or FPGA) were used. This makes a fair comparison of the implementation results almost impossible. The main motivation behind this work is to address this issue by implementing a group of SHA-3 candidates using the same technology and applying the same design approach. We then compare the results of our implementations of the SHA-3 candidates with that of SHA-2.

The SHA-3 candidates we consider here are: Blue Midnight Wish, Luffa, Skein, Shabal and Blake [7]-[11]. These five algorithms are selected since they present faster software implementation results compared to the rest of the proposals and are suitable for hardware implementation [5].

2 Design Approach

In general, most hash algorithms use an iterative process to hash the arbitrary size input by successively processing fixed size blocks of the input [1]. This is achieved through a two stage architecture. The first stage which we refer to as the *Preprocessing stage* is in charge of padding the input to appropriate size (usually multiplies of 256 or 512 bits), and then breaking down the message into blocks of smaller fixed sizes (256 or 512 bits).

Next, the message blocks will be processed by the second stage which is referred to as the *Hash Computation Stage*. In this stage, a special function called the *Compression Function* is used iteratively through a number of rounds to create the message digest. The actual number of rounds (or iterations) depends on the number of message blocks. The first round compression function uses the first block of the message along with a predefined initial value called the *Initialization Vector*. The initialization vector is usually created in the preprocessing stage and is just used for the first round of the compression function. For the rest of the rounds, the outputs of each round compression function will be used as the next round initialization vector.

As NIST has requested, all candidates are capable of supporting different message digest sizes of 224, 256, 384 and 512 bits. However, we have just focused on the 256-bit versions in this work. If the authors of a candidate proposed hardware architectures for their design, we use that for our hardware implementation (Skein, Shabal, and Blake). For Luffa and Blue Midnight Wish the authors did not present any hardware architecture in their proposal, hence their algorithms are implemented as pure combinational circuit. Input/Output registers are added at the input and output ports to take into account the loading effect of the inputs/outputs and also create the register transfer level modules. The inputs for the compression function block is a message of fixed block size (256 or 512 bits according to the algorithm) along with the initialization vector (if needed). The final output of the compression function is a 256 bit message digest.

We present both ASIC and FPGA implementations. For ASIC implementations, VHDL is used to describe the modules in hardware. The code is then simulated using Cadence's NCSim to verify its functionality. Afterwards, the VHDL code is synthesized to a gate-level netlist using Synopsys' Design Compiler. Compilation parameters are always chosen to maximize the operating frequency of the design.

FPGA implementations are carried out using the same VHDL files used for ASIC implementations. The only difference is the addition of the serial-in parallel-out and parallel-in serial-out modules to transfer the data into or out of the FPGA. Note that FPGA implementations take into account the I/O speed limits of the FPGA whereas this limitation is not considered for ASIC implementations. We have used the 90 nm CMOS technology from STMicroelectronics for our ASIC implementations. For FPGA implementations, we use Startix III FPGA from Altera.

We base the comparison of our hardware implementations on area and time requirements of each design. For ASIC implementations, we report total area in μm^2 which includes area due to combinational circuits as well as I/O registers. We also give the total number of gates used for each design. Note that this parameter varies considerably according to the size and types of gates available in the library. For FPGA implementations as we are using Altera products, the area requirements are given in terms of the number of Adaptive Look-Up Tables (ALUTs) and dedicated logic registers.

For time requirements, we report total delay in nano seconds for fully unrolled designs (Luffa, Skein, BMW, and SHA-2) and is equal to the critical path delay. For sequential designs (Skein-1c, Shabal, Blake, and SHA-2-1c) the total delay is the product of the critical path delay and the required number of clock cycles for each iteration.

For FPGA designs, clock signal of the combinational circuit that implements the compression function is typically different from the clock signal of the I/O modules.

3 Blue Midnight Wish Hash Function

3.1 Algorithm Description

The Blue Midnight Wish (BMW) algorithm has been proposed by Svein Johan Knapskog et al. in [7]. We refer to the variant that creates the 256 bit message digest as BMW-256. The basic data block used is called a word which is 32 bits long. BMW makes use of four different operations in the hash computation stage: bit-wise logical word XOR operation, word addition and subtraction (modulo 2^{32}), shift operations (left or right), and rotate left (circular shift left) operation.

BMW uses a *Double Pipe* design to increase the resistance against generic multicollision attacks and length extension attacks [12, 13]. In the double pipe design, the size of the inputs to the compression function are twice the message digest size. The inputs to the compression function are the message blocks M^i of size 512 bits, along with the initialization vector (previous output of the compression function) which is called the old double pipe H^{i-1} of size 512 bits. The final message digest output bits are the least significant 256 bits of the current double pipe H^i.

The compression function uses three separate functions to generate the message digest; f_0, f_1, f_2. The general block diagram for the BMW-256 compression function is shown in Fig. 1. Note that each of the signals in the figure is 512 bits wide.

The first function $f_0 : \{0,1\}^{2*512} \longrightarrow \{0,1\}^{512}$ takes two arguments; a message block M^i and the previous value of the double pipe H^{i-1}, and generates

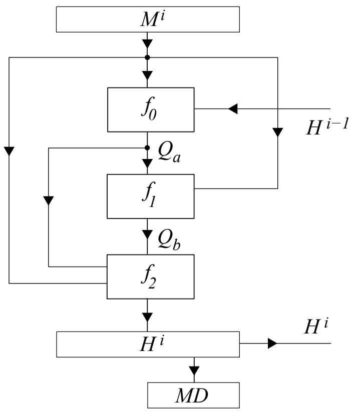

Fig. 1. General block diagram of the compression function in BMW-256

$Q_a^i = Q^i_{(0,1,\ldots,15)}$ the first part of the quadruple pipe of size 512 bits. Mathematical definition for the function f_0 is shown in Eqs. (1)-(3).

In these equations, initially temporary signal $W^i_{(0,1,\ldots,15)}$ is generated through word by word XOR of the two inputs and addition/subtraction of the words. Note that all addition and subtractions are done module 2^{32}, hence carry propagation just exists inside the words. Signal W is then used to create the Q_a signal using s_0, s_1, s_2, s_3, s_4 *s-transforms* according to Eq. (2). Each transformation accepts a 32-bit word and generates another 32-bit word. Details of *s-transforms* are shown in Eq. (3). Note that in this equation $SHR^r(x)/SHL^r(x)$ represents the shift right/left operation by r bits, while $ROTL^r(x)$ represents the rotate left operation by r bits.

$$
\begin{aligned}
W_0^{(i)} &= (M_5^{(i)} \oplus H_5^{(i-1)}) - (M_7^{(i)} \oplus H_7^{(i-1)}) + (M_{10}^{(i)} \oplus H_{10}^{(i-1)}) + (M_{13}^{(i)} \oplus H_{13}^{(i-1)}) + (M_{14}^{(i)} \oplus H_{14}^{(i-1)})\\
W_1^{(i)} &= (M_6^{(i)} \oplus H_6^{(i-1)}) - (M_8^{(i)} \oplus H_8^{(i-1)}) + (M_{11}^{(i)} \oplus H_{11}^{(i-1)}) + (M_{14}^{(i)} \oplus H_{14}^{(i-1)}) - (M_{15}^{(i)} \oplus H_{15}^{(i-1)})\\
W_2^{(i)} &= (M_0^{(i)} \oplus H_0^{(i-1)}) + (M_7^{(i)} \oplus H_7^{(i-1)}) + (M_9^{(i)} \oplus H_9^{(i-1)}) - (M_{12}^{(i)} \oplus H_{12}^{(i-1)}) + (M_{15}^{(i)} \oplus H_{15}^{(i-1)})\\
W_3^{(i)} &= (M_0^{(i)} \oplus H_0^{(i-1)}) - (M_1^{(i)} \oplus H_1^{(i-1)}) + (M_8^{(i)} \oplus H_8^{(i-1)}) - (M_{10}^{(i)} \oplus H_{10}^{(i-1)}) + (M_{13}^{(i)} \oplus H_{13}^{(i-1)})\\
W_4^{(i)} &= (M_1^{(i)} \oplus H_1^{(i-1)}) + (M_2^{(i)} \oplus H_2^{(i-1)}) + (M_9^{(i)} \oplus H_9^{(i-1)}) - (M_{11}^{(i)} \oplus H_{11}^{(i-1)}) - (M_{14}^{(i)} \oplus H_{14}^{(i-1)})\\
W_5^{(i)} &= (M_3^{(i)} \oplus H_3^{(i-1)}) - (M_2^{(i)} \oplus H_2^{(i-1)}) + (M_{10}^{(i)} \oplus H_{10}^{(i-1)}) - (M_{12}^{(i)} \oplus H_{12}^{(i-1)}) + (M_{15}^{(i)} \oplus H_{15}^{(i-1)})\\
W_6^{(i)} &= (M_4^{(i)} \oplus H_4^{(i-1)}) - (M_0^{(i)} \oplus H_0^{(i-1)}) - (M_3^{(i)} \oplus H_3^{(i-1)}) - (M_{11}^{(i)} \oplus H_{11}^{(i-1)}) + (M_{13}^{(i)} \oplus H_{13}^{(i-1)})\\
W_7^{(i)} &= (M_1^{(i)} \oplus H_1^{(i-1)}) - (M_4^{(i)} \oplus H_4^{(i-1)}) - (M_5^{(i)} \oplus H_5^{(i-1)}) - (M_{12}^{(i)} \oplus H_{12}^{(i-1)}) - (M_{14}^{(i)} \oplus H_{14}^{(i-1)})\\
W_8^{(i)} &= (M_2^{(i)} \oplus H_2^{(i-1)}) - (M_5^{(i)} \oplus H_5^{(i-1)}) - (M_6^{(i)} \oplus H_6^{(i-1)}) + (M_{13}^{(i)} \oplus H_{13}^{(i-1)}) - (M_{15}^{(i)} \oplus H_{15}^{(i-1)})\\
W_9^{(i)} &= (M_0^{(i)} \oplus H_0^{(i-1)}) - (M_3^{(i)} \oplus H_3^{(i-1)}) + (M_6^{(i)} \oplus H_6^{(i-1)}) - (M_7^{(i)} \oplus H_7^{(i-1)}) + (M_{14}^{(i)} \oplus H_{14}^{(i-1)})\\
W_{10}^{(i)} &= (M_8^{(i)} \oplus H_8^{(i-1)}) - (M_1^{(i)} \oplus H_1^{(i-1)}) - (M_4^{(i)} \oplus H_4^{(i-1)}) - (M_7^{(i)} \oplus H_7^{(i-1)}) + (M_{15}^{(i)} \oplus H_{15}^{(i-1)})\\
W_{11}^{(i)} &= (M_8^{(i)} \oplus H_8^{(i-1)}) - (M_0^{(i)} \oplus H_0^{(i-1)}) - (M_2^{(i)} \oplus H_2^{(i-1)}) - (M_5^{(i)} \oplus H_5^{(i-1)}) + (M_9^{(i)} \oplus H_9^{(i-1)})\\
W_{12}^{(i)} &= (M_1^{(i)} \oplus H_1^{(i-1)}) + (M_3^{(i)} \oplus H_3^{(i-1)}) - (M_6^{(i)} \oplus H_6^{(i-1)}) - (M_9^{(i)} \oplus H_9^{(i-1)}) + (M_{10}^{(i)} \oplus H_{10}^{(i-1)})\\
W_{13}^{(i)} &= (M_2^{(i)} \oplus H_2^{(i-1)}) + (M_4^{(i)} \oplus H_4^{(i-1)}) + (M_7^{(i)} \oplus H_7^{(i-1)}) + (M_{10}^{(i)} \oplus H_{10}^{(i-1)}) + (M_{11}^{(i)} \oplus H_{11}^{(i-1)})\\
W_{14}^{(i)} &= (M_3^{(i)} \oplus H_3^{(i-1)}) - (M_5^{(i)} \oplus H_5^{(i-1)}) + (M_8^{(i)} \oplus H_8^{(i-1)}) - (M_{11}^{(i)} \oplus H_{11}^{(i-1)}) - (M_{12}^{(i)} \oplus H_{12}^{(i-1)})\\
W_{15}^{(i)} &= (M_{12}^{(i)} \oplus H_{12}^{(i-1)}) - (M_4^{(i)} \oplus H_4^{(i-1)}) - (M_6^{(i)} \oplus H_6^{(i-1)}) - (M_9^{(i)} \oplus H_9^{(i-1)}) + (M_{13}^{(i)} \oplus H_{13}^{(i-1)})
\end{aligned}
\tag{1}
$$

$$
\begin{gathered}
Q_0^i = s_0(W_0^i), \quad Q_1^i = s_1(W_1^i), \quad Q_2^i = s_2(W_2^i), \quad Q_3^i = s_3(W_3^i), \quad Q_4^i = s_4(W_4^i), \quad Q_5^i = s_0(W_5^i),\\
Q_6^i = s_1(W_6^i), \quad Q_7^i = s_2(W_7^i), \quad Q_8^i = s_3(W_8^i), \quad Q_9^i = s_4(W_9^i), \quad Q_{10}^i = s_0(W_{10}^i), \quad Q_{11}^i = s_1(W_{11}^i),\\
Q_{12}^i = s_2(W_{12}^i), \quad Q_{13}^i = s_3(W_{13}^i), Q_{14}^i = s_4(W_{14}^i), \quad Q_{15}^i = s_0(W_{15}^i),
\end{gathered}
\tag{2}
$$

$$
\begin{aligned}
s_0(x) &= SHR^1(x) \oplus SHL^3(x) \oplus ROTL^4(x) \oplus ROTL^{19}(x) \quad s_4(x) = SHR^1(x) \oplus x\\
s_1(x) &= SHR^1(x) \oplus SHL^2(x) \oplus ROTL^8(x) \oplus ROTL^{23}(x) \quad s_5(x) = SHR^2(x) \oplus x\\
s_2(x) &= SHR^2(x) \oplus SHL^1(x) \oplus ROTL^{12}(x) \oplus ROTL^{25}(x)\\
s_3(x) &= SHR^2(x) \oplus SHL^2(x) \oplus ROTL^{15}(x) \oplus ROTL^{29}(x)
\end{aligned}
\tag{3}
$$

The second function $f_1 : \{0,1\}^{2*512} \longrightarrow \{0,1\}^{512}$ takes the same message block M^i as the first function and the output of the first function $Q_a^i = Q_{(0,1,\dots,15)}^i$ and generates the second part of the quadruple pipe $Q_b^i = Q_{(16,17,\dots,31)}^i$, through EXPAND1 and EXPAND2 expansion functions as shown in Eq. (4). Expansion functions make use of the *s-transforms* along with the *r-transforms* which are defined in Eq. (5).

$$\begin{aligned}
&For \quad ii = 0,1 \quad : \quad Q^i_{(ii+16)} = Expand1(ii+16)\\
&For \quad ii = 2\,to\,15 \quad : \quad Q^i_{(ii+16)} = Expand2(ii+16)
\end{aligned}$$

$$\begin{aligned}
Expand1(j) &= s_1(Q^i_{(j-16)}) + s_2(Q^i_{(j-15)}) + s_3(Q^i_{(j-14)}) + s_0(Q^i_{(j-13)})\\
&+ s_1(Q^i_{(j-12)}) + s_2(Q^i_{(j-11)}) + s_3(Q^i_{(j-10)}) + s_0(Q^i_{(j-9)})\\
&+ s_1(Q^i_{(j-8)}) + s_2(Q^i_{(j-7)}) + s_3(Q^i_{(j-6)}) + s_0(Q^i_{(j-5)})\\
&+ s_1(Q^i_{(j-4)}) + s_2(Q^i_{(j-3)}) + s_3(Q^i_{(j-2)}) + s_0(Q^i_{(j-1)})\\
&+ M^i_{(j-16)mod16} + M^i_{(j-13)mod16} - M^i_{(j-6)mod16} + j \times 0x05555555\\
Expand2(j) &= Q^i_{(j-16)} + r_1(Q^i_{(j-15)}) + Q^i_{(j-14)} + r_2(Q^i_{(j-13)})\\
&+ Q^i_{(j-12)} + r_3(Q^i_{(j-11)}) + Q^i_{(j-10)} + r_4(Q^i_{(j-9)})\\
&+ Q^i_{(j-8)} + r_5(Q^i_{(j-7)}) + Q^i_{(j-6)} + r_6(Q^i_{(j-5)})\\
&+ Q^i_{(j-4)} + r_7(Q^i_{(j-3)}) + Q^i_{(j-2)} + r_4(Q^i_{(j-1)})\\
&+ M^i_{(j-16)mod16} + M^i_{(j-13)mod16} - M^i_{(j-6)mod16} + j \times 0x05555555
\end{aligned} \tag{4}$$

$$\begin{aligned}
&r_1(x) = ROL^3(x),\ \ r_2 = ROTL^7(x),\ \ r_3(x) = ROTL^{13}(x)\\
&r_4(x) = ROL^{16}(x),\ r_5 = ROTL^{19}(x),\ r_6(x) = ROTL^{23}(x)\\
&r_7(x) = ROL^{27}(x),
\end{aligned} \tag{5}$$

Finally, the last function $f_2 : \{0,1\}^{3*512} \longrightarrow \{0,1\}^{512}$ takes two arguments; the message block M^i, and the quadruple pipe $Q^{i-1}_{0,1,\ldots,31} = Q^i_a \,||\, Q^i_b$, ($||$ represents the concatenation operation) and generates the current value of the double pipe H^i. Mathematical definition for the function f_2 is shown in Eq. (6).

$$
\begin{aligned}
XL &= Q^i_{16} \oplus Q^i_{17} \oplus Q^i_{18} \oplus Q^i_{19} \oplus Q^i_{20} \oplus Q^i_{21} \oplus Q^i_{22} \oplus Q^i_{23} \\
XL &= XL \oplus Q^i_{24} \oplus Q^i_{25} \oplus Q^i_{26} \oplus Q^i_{27} \oplus Q^i_{28} \oplus Q^i_{29} \oplus Q^i_{30} \oplus Q^i_{31} \\
\\
H^i_0 &= (SHL^5(XH) \oplus SHR^5(Q^i_{16} \oplus M^i_0) + (XL \oplus Q^i_{24} \oplus Q^i_0) \\
H^i_1 &= (SHR^7(XH) \oplus SHL^8(Q^i_{17} \oplus M^i_1) + (XL \oplus Q^i_{25} \oplus Q^i_1) \\
H^i_2 &= (SHR^5(XH) \oplus SHL^5(Q^i_{18} \oplus M^i_2) + (XL \oplus Q^i_{26} \oplus Q^i_2) \\
H^i_3 &= (SHR^1(XH) \oplus SHL^5(Q^i_{19} \oplus M^i_3) + (XL \oplus Q^i_{27} \oplus Q^i_3) \\
H^i_4 &= (SHR^3(XH) \oplus Q^i_{20} \oplus M^i_4) + (XL \oplus Q^i_{28} \oplus Q^i_4) \\
H^i_5 &= (SHL^6(XH) \oplus SHR^6(Q^i_{21} \oplus M^i_5) + (XL \oplus Q^i_{29} \oplus Q^i_5) \\
H^i_6 &= (SHR^4(XH) \oplus SHLhQ^i_{22} \oplus M^i_6) + (XL \oplus Q^i_{30} \oplus Q^i_6) \\
H^i_7 &= (SHR^{11}(XH) \oplus SHL^2(Q^i_{23} \oplus M^i_7) + (XL \oplus Q^i_{31} \oplus Q^i_7) \\
H^i_8 &= ROTL^9(H^i_4) + (XH \oplus Q^i_{24} \oplus M^i_8) + (SHL^8(XL) \oplus Q^i_{23} \oplus Q^i_8) \\
H^i_9 &= ROTL^{10}(H^i_5) + (XH \oplus Q^i_{25} \oplus M^i_9) + (SHR^6(XL) \oplus Q^i_{16} \oplus Q^i_9) \\
H^i_{10} &= ROTL^{11}(H^i_6) + (XH \oplus Q^i_{26} \oplus M^i_{10}) + (SHL^6(XL) \oplus Q^i_{17} \oplus Q^i_{10}) \\
H^i_{11} &= ROTL^{12}(H^i_7) + (XH \oplus Q^i_{27} \oplus M^i_{11}) + (SHL^4(XL) \oplus Q^i_{18} \oplus Q^i_{11}) \\
H^i_{12} &= ROTL^{13}(H^i_0) + (XH \oplus Q^i_{28} \oplus M^i_{12}) + (SHR^3(XL) \oplus Q^i_{19} \oplus Q^i_{12}) \\
H^i_{13} &= ROTL^{14}(H^i_1) + (XH \oplus Q^i_{29} \oplus M^i_{13}) + (SHR^4(XL) \oplus Q^i_{20} \oplus Q^i_{13}) \\
H^i_{14} &= ROTL^{15}(H^i_2) + (XH \oplus Q^i_{30} \oplus M^i_{14}) + (SHR^7(XL) \oplus Q^i_{21} \oplus Q^i_{14}) \\
H^i_{15} &= ROTL^{16}(H^i_3) + (XH \oplus Q^i_{31} \oplus M^i_{15}) + (SHR^2(XL) \oplus Q^i_{22} \oplus Q^i_{15})
\end{aligned}
\tag{6}
$$

3.2 ASIC Implementation

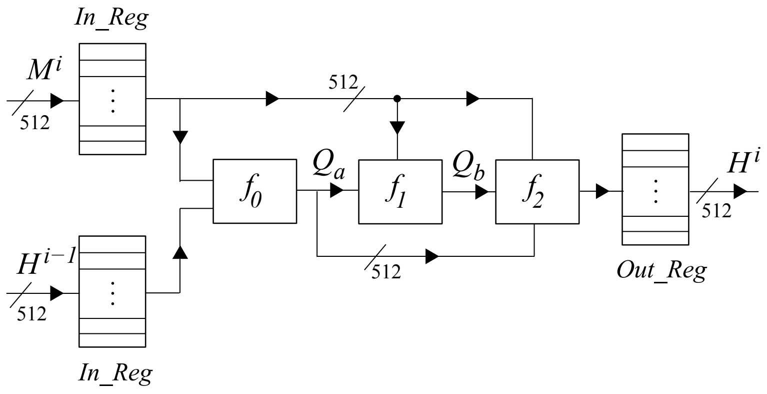

Fig. 2. hardware architecture used for the compression function of the BMW-256

Block diagram of the architecture that is used for the ASIC implementation of the BMW-256 is shown in Fig. 2. In this figure, each line presents a 512-bit wide signal. Inputs M^i and H^{i-1} are stored inside the 512-bit input registers

(In_Reg). Functions f_1, f_2, f_3 were realized completely as combinational circuits. Output H^i can be read from the output register (Out_Reg) after one clock cycle.

Input Size	Compression Function Delay	Combinational Area	I/O Register Area	Number of Gates
512	$19.20\,ns$	$670,887.14\,\mu m^2$	$48,746.61\,\mu m^2$	$60,033$

Table 1. ASIC implementation summary of the BMW-256 compression function

Synthesis results for the BMW-256 compression function are summarized in Table 1. In this table, Compression Function Delay represents the maximum delay between the Input and Output registers. Combinational Area represents the area used by functions f_0, f_1, and f_2. I/O Register Area shows the area utilized by the In_reg and Out_reg registers. Number of gates may be considered to be an alternative to measure the area requirements of the implementation.

4 Luffa Hash Function

4.1 Algorithm Description

The Luffa algorithm s due to Christophe De Canniere et al. in [8]. It makes use of s-boxes (non-linear permutation) along with shift, and XOR operations to create the output message digest. The basic data block used is a word which is 32 bits long. Luffa's compression function is called the round function. The structure of a round function for Luffa-256 is shown in Fig. 3. It is made of two separate functions; Message Injection (MI) and Permutation (P).

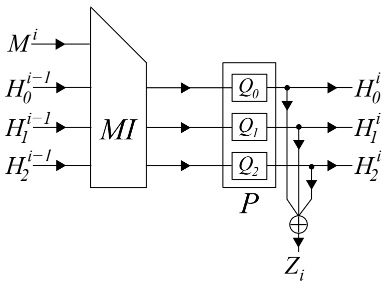

Fig. 3. General block diagram of the compression function (round function) for Luffa-256

The detail of the message injection module is shown in Fig. 4. In this figure the circled plus sign ($\oplus$) represents bit by bit XOR operation of the variables (256 bit each). Also the circled cross sign ($\otimes$) represents the multiplication operation in the ring of $GF(2^8)^{32}$ using the polynomial $\phi(x) = x^8 + x^4 + x^3 + x + 1$.

The multiplication operation can be defined as follows; Assume that A and B are each a vector of 256 bits and $A = a[7]||...||a[0]$ and $B = b[7]||...||b[0]$ where $a[i], b[i], i = 0, ..., 7$ are vectors of 32 bits each. Assume that $B = A \otimes 2$, then we have

$$\begin{array}{lll} b[7] = a[6], & b[6] = a[5], & b[5] = a[4] \\ b[4] = a[3] XOR a[7], & b[3] = a[2] XOR a[7], & b[2] = a[1] \\ b[1] = a[0] XOR a[7], & b[0] = a[7] & \end{array} \tag{7}$$

The permutation stage (P) for Luffa-256 is made of three similar permutation blocks $Q_j, j = 0, 1, 2$ which work in parallel, as shown in Fig. 3. We refer to these blocks as *Permute* blocks whose input and output sizes are 256 bits each. They can be modelled as a composition of an input tweak and eight iterations of an Step function. The tweak module is just reordering of the wires and does not contain any gates. The block diagram of the Step function is shown in Fig. 5.

In this figure, the 256-bit input to the step function is stored in eight 32-bit registers denoted by $a_k, 0 \leqslant k < 8$ where k represents the register number. Three main submodules exist in the step function: SubCrumb, MixWord, and the AddConstant. The SubCrumb module is a nonlinear permutation which makes use of 32 similar s-boxes (4-bit input, 4- bit output) in parallel for its operation. The s-box input/output relationship is as follows.

$$s[16] = \{7, 13, 11, 10, 12, 4, 8, 3, 5, 15, 6, 0, 9, 1, 2, 14\} \tag{8}$$

The MixWord is a linear permutation module which mixes two words together. The structure of the MixWord is shown in Fig. 6.

In Fig. 5, the AddConstant module performs two XOR operations involving constants according to the parameters k and r (the Step number) to the first and the fourth words. Note that since the value of the constants are predetermined (one input of the XOR gates), the synthesizer will perform optimization and replace the XOR gates by the Inverters or wires as required.

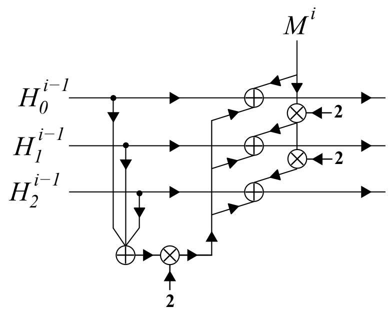

Fig. 4. Block diagram of the Message Injection module for Luffa-256

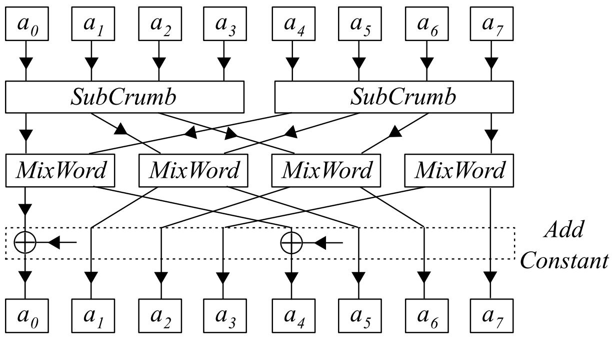

Fig. 5. Block diagram of the Step function module for Luffa-256

4.2 ASIC Implementation

Block diagram of the architecture that is used for the ASIC implementation of Luffa-256 is shown in Fig. 7. In this figure, each line presents a 256-bit wide signal.

Inputs M^i and H^{i-1} are stored inside the 256-bit input registers (In_Reg). MI and P blocks (including the s-boxes) are realized completely as combinational circuits. Output H^i can be read from the output register (Out_Reg) after one clock cycle. Synthesis results for the Luffa-256 compression function are summarized in Table 2. In this table, the Combinational Area represents the area used by the message injection and permutation (three permute blocks). Each permute block contains a tweak module and eight instances of the step function.

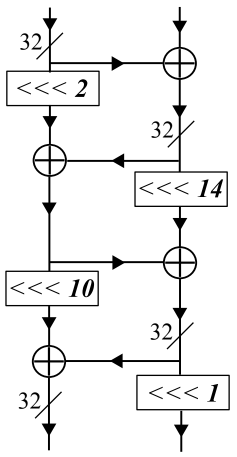

Fig. 6. Block diagram of the MixWord function module for Luffa-256

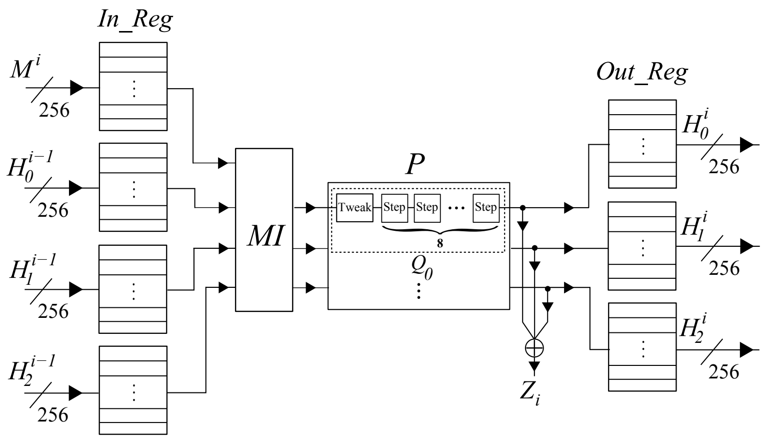

Fig. 7. Hardware architecture used for the compression function of Luffa-256

5 Skein Hash Function

5.1 Algorithm Description

Ferguson et al. have proposed the Skein algorithm [9]. Skein makes use of 64-bit adders along with shift, and XOR operations to create the output message digest. The basic data block used is a word which is 64 bits long. Skein's compression function is based on *Threefish* which is a large tweakable block cipher [14]. Skein-256 compression function is made of 72 consecutive rounds of Threefish after four of each there exist a subkey addition module. Subkeys are generated form an input key through the key schedule module. Note that Skein does not make use of the initialization vector the way Luffa and Blue Midnight Wish do.

Skein is built on multiple invocations of the Unique Block Iteration (UBI) which is a variant of the Matyas-Meyer-Oseas hash mode [15]. The basic building block of UBI is shown in Fig. 8.

The structure of the Skein compression function (including the subkey modules) is shown in Fig. 9. Note that the key is not an optional parameter for Threefish and can not be removed from the Skein structure. subkeys are consist of three contributions: key words, tweak words, and a counter value.

Each round of the Threefish block cipher (256 bit version) is made of two instances of a Mix function along with a permutation module. The structure of the Mix function for Threefish-256 is shown in Fig. 10. In this figure the boxed

Input Size	Compression Function Delay	Combinational Area	I/O Register Area	Number of Gates
256	$9.96\,ns$	$473,318.04\,\mu m^2$	$60,277.99\,\mu m^2$	$68,884$

Table 2. ASIC implementation summary of the Luffa-256 compression function

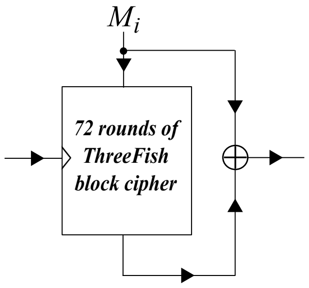

Fig. 8. Unique Block Iteration (UBI) building block

plus sign (⊞) represents a 64 bit addition (modulo 2^{64}), and the circled plus sign (⊕) represents a bit by bit XOR operation. Parameters $R_{d,i}$ are constants that set the size of the rotate shift operations according to the round number.

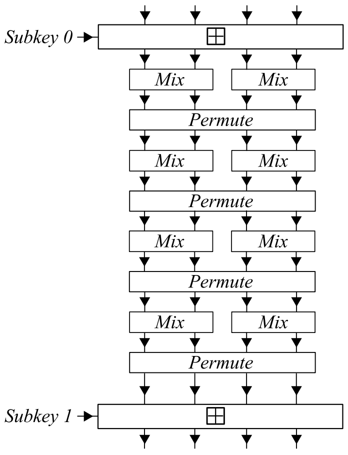

Fig. 9. Block diagram of compression function module for Skein-256, 4 out of 72 rounds for Skein-256 including the key

5.2 ASIC Implementation

Fully Unrolled/Parallel Design

Block diagram of the architecture used for the hardware implementation of Skein-256 is shown in Fig. 11. In this figure, each line presents a 64-bit wide signal except the input/output wires which are 256 bits wide and the Tweak and key input wires which are 128 and 256 bits respectively.

Input Size	Compression Function Delay	Combinational Area	I/O Register Area	Number of Gates
256	$81.92\,ns$	$1,605,827.19\,\mu m^2$	$13,788.05\,\mu m^2$	$252,725$

Table 3. ASIC implementation summary of the Skein-256 compression function

Input message M^i is stored inside the 256-bit input registers (In_Reg). The Mix functions have been realized completely as combinational circuits. After 72 rounds of Mix and Permute functions (including 18 rounds of key addition), one level of XOR gates exist to create the UBI structure of the algorithm. Output H^i can be read from the output register (Out_Reg) after one clock cycle. Synthesis results of the Skein-256 compression function are summarized in Table 3. In this table, the Combinational Area represents the area used by the 144 Mix functions and 18 subkey adder modules exist in 72 rounds of Threefish along with 256 XOR gates in Skein-256.

Sequential Design

Skein compression function architecture presents a highly regular design as shown in Fig. 9. This property can be exploited to create a complete Skein compression function by just implementing one round of the Threefish cipher and a subkey adder module. This can play a major rule when the hashing algorithm is implemented for area constrained applications. To this end, we have implemented a second version of the Skein algorithm (Skein-1c) using just one round of the Threefish algorithm.

Note that some modifications to the original round function of Threefish have been allowed to make the design of Skein-1c capable of creating the message digest. The main modification here is the addition of program-ability option to the Mix module, since the rotate shift operation is a function of the round number. Also some extra modules (a counter and two multiplexers and a key schedule module) have been used to create the correct inputs for the round function as shown in Fig. 12. The key schedule module uses circular shift registers along with three 64-bit adders to generate the subkeys.

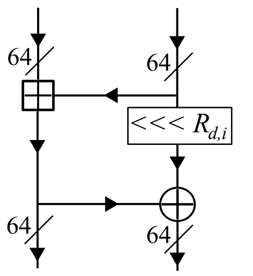

Fig. 10. Block diagram of the Mix function module for Skein-256

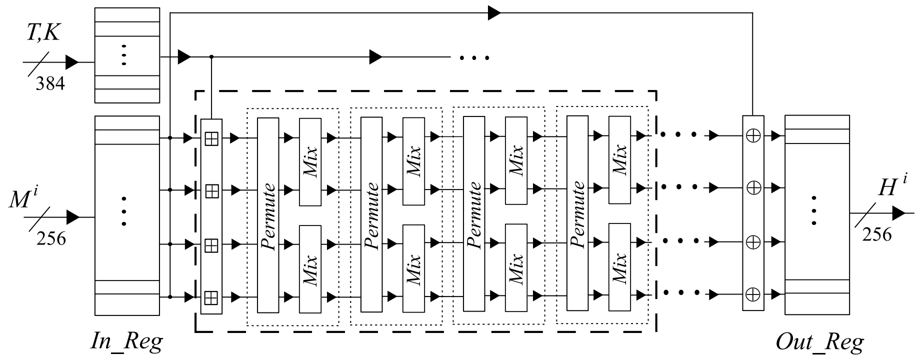

Fig. 11. Hardware architecture used for the compression function of Skein-256

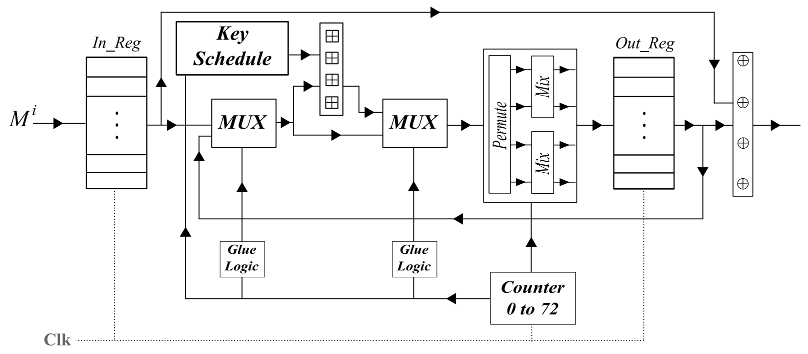

Fig. 12. Hardware architecture used for the compression function of Skein-1c

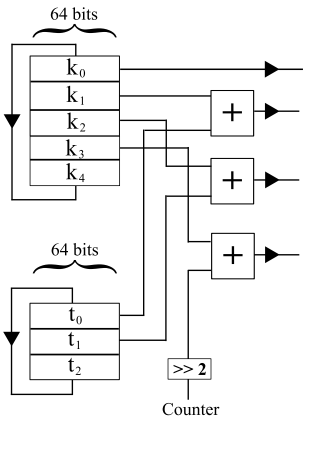

Fig. 13. Hardware architecture used for the key schedule of Skein-1c

Block diagram of the architecture that is used for key schedule is shown in Fig. 12. It should be noted that the clock used for the key module is four times slower than the clock used for the rest of the Skein-1c design. Also we had assumed that the two parameters k_4 and t_2 are available at the start of the circuit operation and are loaded into the circular shift registers.

The design is therefore referred to as Skein-1c since it uses one round of the Threefish compression function and it also includes an extra counter. Synthesis results for skein-1c compression function are summarized in Table 4.

Input Size	Critical Path Delay	Number of Cycles	Combinational Area	I/O Register Area	Number of Gates
256	$3.49\,ns$	72	$64,809.98\,\mu m^2$	$27,927.33\,\mu m^2$	$11,634$

Table 4. ASIC implementation summary of the Skein-1c compression function

From Tables 3 and 4 the following can be deduced: the area requirement for the Skein compression function has been reduced from $1,619,615.24\,\mu m^2$ in Skein-256 to $92,737.32\,\mu m^2$ in Skein-1c. This reduction comes at the price of increasing the compression function delay from $81.92\,ns$ in Skein-256 to $251.28\,ns$ in Skein-1c.

6 Shabal Hash Function

6.1 Algorithm Description

The Shabal algorithm has been proposed by Bresson et al. in [10]. Shabal architecture is based on Linear Feedback Shift Register (LFSR), meaning the inputs to the current state of the registers are linear function of the previous state of the registers. Shabal makes use of 32-bit adders/subtractors along with shift, and XOR operations to create the output message digest. The basic data block used is a word which is 32 bits long.

The simplified structure of the compression function block is shown in Fig. 14. In this figure, the boxed plus sign ($\boxplus$) represents a 32 bit addition (modulo 2^{32}), and the boxed minus sign ($\boxminus$) represents a 32 bit subtraction. Circled plus sign ($\oplus$) represents a bit by bit XOR operation. A, B and C are initialization vectors, while W is a 64 bit counter used to number the message blocks. The P block represents a keyed permutation module and its details are shown in figure 15.

In Fig. 15, inputs B, C, and M are represented by sixteen words each. The 32-bit registers are shown as one block in the figure. Note that input A contains only twelve words and during each clock cycle one word (32 bits) is shifted towards left or right according to the architecture. In this figure modules U and V are non linear functions defined by:

$U : x \mapsto 3 \times x \ mod\ 2^{32}$ and $v : x \mapsto 5 \times x \ mod\ 2^{32}$. In hardware, they can be implemented as small constant multipliers or as two additions and shift for U and three additions and shift for V. Also symbol $\oplus$ represents a bit by bit

XOR operation. Note that the $\oplus$ module having $0xFFF...F$ as one of its inputs, represents an inversion operation.

6.2 ASIC Implementation

We have used the same block diagrams as Fig. 14 and Fig. 15 for hardware implementation of Shabal-256. The only modification applied is the addition of a way to load inputs A,B,C and M into the keyed permutation module. This is achieved by adding a small multiplexer (having one select line and two inputs) to each register storing the inputs. This way all input bits could be loaded into the module, similar to other architectures that have been used in our hardware implementations. Synthesis results for the Shabal-256 compression function are summarized in Table 5.

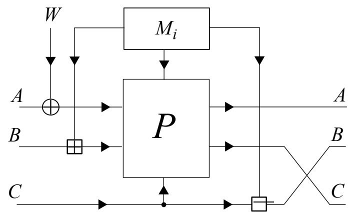

Fig. 14. Block diagram of compression function for Shabal-256

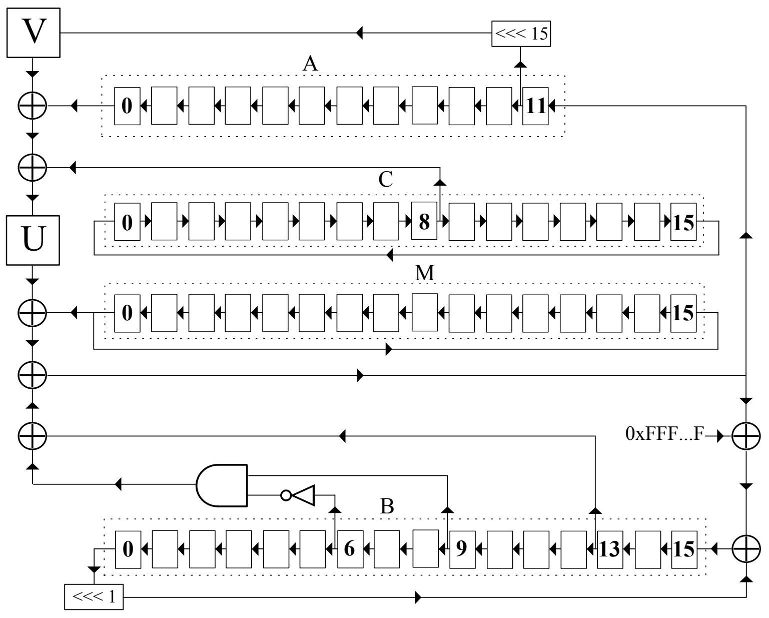

Fig. 15. Structure of the keyed permutation module P

Input Size	Critical Path Delay	Number of Cycles	Combinational Area	I/O Register Area	Number of Gates
256	$2.42\,ns$	16	$38,244.77\,\mu m^2$	$49,041.86\,\mu m^2$	$8,065$

Table 5. ASIC implementation summary of the Shabal-256 compression function

In this table, the Combinational Area represents the area used by U and V modules along with the area used by the XOR gates and multiplexers. Major part of the area usage belongs to the five 32-bit adders realizing modules U and V.

7 Blake Hash Function

7.1 Algorithm Description

The Blake algorithm has been proposed by Aumasson et al. in [11]. The basic data block used is called a word which is 32 bits long. Blake makes use of bit-wise logical word XOR operations, word addition (modulo 2^{32}), rotate operations (left or right), and permutations in its structure.

Blake is mainly built on previously designed components. Its iteration mode is HAIFA which is an improved version of Merkle-Damgard [16]. Its compression function is a modified version of the ChaCha stream cipher [17]. It also supports an optional salt (a parameter that can be either public or secret) to create the message digest. Blake uses a *Wide Pipe* design to increase its resistance against collision attacks [18, 19]. The local wide pipe structure of Blake's compression function is shown in Fig. 16.

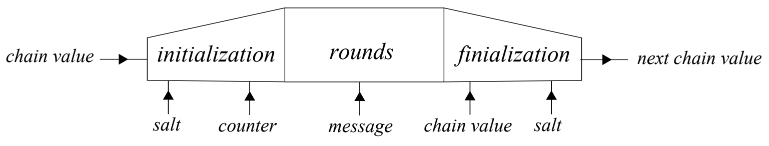

Fig. 16. Block diagram of compression function for Blake-256

As can be seen from the figure, the compression function is made of three main stages: initialization, rounds, and finalization. In the initialization stage a large inner state is initialized from the previous chain value (h), salt (t), and counter (t). The state value is then updated by message-dependent rounds. It is finally compressed to create the next chain value. A sixteen word state $v_0, v_1, \cdots, v_{15}$ is generated in the initialization stage. The input/output relationship of the initialization step can be mathematically represented as follows.

$$\begin{bmatrix} v_0 & v_1 & v_2 & v_3 \\ v_4 & v_5 & v_6 & v_7 \\ v_8 & v_9 & v_{10} & v_{11} \\ v_{12} & v_{13} & v_{14} & v_{15} \end{bmatrix} \leftarrow \begin{bmatrix} h_0 & h_1 & h_2 & h_3 \\ h_4 & h_5 & h_6 & h_7 \\ s_0 \oplus c_0 & s_1 \oplus c_1 & s_2 \oplus c_2 & s_3 \oplus c_3 \\ t_0 \oplus c_4 & t_0 \oplus c_5 & t_1 \oplus c_6 & t_1 \oplus c_7 \end{bmatrix} \tag{9}$$

Ten rounds of transformation are applied in the rounds stage, after the state v is initialized. Each round of transformation is made of eight operations using the G_i functions defined as follows.

$$\begin{array}{llll} G_0(v_0, v_4, v_8, v_{12}) & G_1(v_1, v_5, v_9, v_{13}) & G_2(v_2, v_6, v_{10}, v_{14}) & G_3(v_3, v_7, v_{11}, v_{15}) \\ G_4(v_0, v_5, v_{10}, v_{15}) & G_5(v_1, v_6, v_{11}, v_{12}) & G_6(v_2, v_7, v_8, v_{13}) & G_7(v_3, v_4, v_9, v_{14}) \end{array} \tag{10}$$

Note that the first four G functions (G_0, G_1, G_2, G_3) can be computed in parallel since they apply to different set of inputs i.e., G_0 applies to v_0, v_4, v_8, v_{12}. Once the first four G functions are applied, the last four (G_4, G_5, G_6, G_7) can be applied. If desirable the last four G functions can be computed in parallel as well. A mathematical definition for function $G_i(a, b, c, d)$ is shown below.

$$\begin{aligned} & a \leftarrow a + b + \big(m_{\sigma_r}(2i) \oplus c_{\sigma_r}(2i+1)\big) \\ & d \leftarrow (d \oplus a) \lll 16 \\ & c \leftarrow c + d \\ & b \leftarrow (b \oplus c) \lll 12 \\ & a \leftarrow a + b + \big(m_{\sigma_r}(2i+1) \oplus c_{\sigma_r}(2i)\big) \\ & d \leftarrow (d \oplus a) \lll 8 \\ & c \leftarrow c + d \\ & b \leftarrow (b \oplus c) \lll 7 \end{aligned} \tag{11}$$

In this equation, c_is are 32-bit word constants and σ_is are permutations of $\{0, 1, \ldots, 15\}$, both defined in the Blake submission documents [11]. Architecture used for the hardware implementation of the G_i function is shown in Fig. 17.

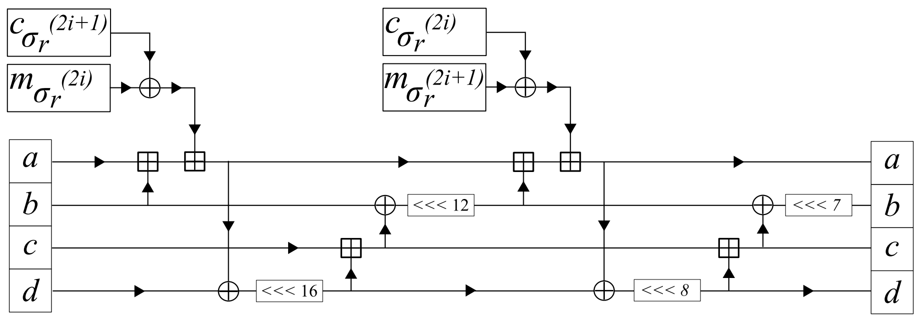

Fig. 17. Block diagram of the G_i function for Blake-256

After 10 rounds of transformation, the next chain value $(h'_0, h'_1, \cdots, h'_7)$ are extracted from the state values $(v_0, v_1, \cdots, v_{15})$ in the finalization stage. The input/output relationship of the finalization step can be mathematically represented as follows.

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

7.2 ASIC Implementation

For our hardware implementation we have used the VHDL code available with the initial submission of Blake. The only modification applied is the addition of the input/output registers to take into account the loading effect on the inputs/outputs. Synthesis results for the Blake-256 compression function are summarized in Table 6.

Input Size	Critical Path Delay	Number of Cycles	Combinational Area	I/O Register Area	Number of Gates
512	$10.4\,ns$	11	$191,408.26\,\mu m^2$	$42,204.91\,\mu m^2$	$30,262$

Table 6. ASIC implementation summary of the Blake-256 compression function

8 Comparison of ASIC Implementation Results

As mentioned earlier, for our implementations we have used the 90 nm CMOS technology from STMicroelecronics; synthesis parameters have always been chosen to maximize the operating frequency of the designs.

Area/Delay complexities of ASIC implementations of all of the SHA-3 candidates considered in this work are listed in Table 7. For simple comparison, the table also includes our two implementations of SHA-2 as described in Appendix A. At the bottom of the table, SHA-2 presents a high-speed 64-round implementation of the SHA-256 architecture, while SHA-2-1c presents a low-area 1-round implementation.

One should note that even though all listed algorithms create the message digest of size 256 bits, their input block size is different. BMW, Blake, SHA-2 and SHA-2-1c use input sizes of 512 bits while Luffa, Skein, Skein-1c and Shabal each accept inputs of size 256 bits. This will make a difference in delay comparisons if the input message size is small (< 256) or very large ($>> 512$).

From Table 7, we can see that Luffa-256 presents the fastest architecture (9.96 ns) followed by Blue Midnight Wish (19.20 ns) and Shabal (38.72 ns). Note that for very large input message sizes (>> 512) BMW performs slightly better than Luffa, and Shabal still ranks third.

Luffa's high speed comes from the fact that it does not make use of any adders (32 bit or 64 bit) in the algorithm. The most expensive building block in the Luffa algorithm is the SubCrumb module (s-box) which can be implemented with 5-6 levels of logic gates. The major part of the delay in Blue Midnight Wish and Shabal comes from the 32 bit adder/subtractor modules used in their architectures.

Skein-1c and Shabal architectures present the smallest critical path delay in our comparison table; however a number of clock cycles are required for these two designs to finish one round of compression (72 and 16 clock cycles). This results in architectures with low performance with respect to the total compression time. However these two architectures present the smallest area utilization, approximately 0.092 mm^2 for Skein-1c and $\simeq 0.087\, mm^2$ for Shabal. Compared to SHA-2-1c both designs require larger area while Skein-1c presents a slower design and Shabal presents a faster one.

Fully unrolled Skein requires the highest area among all proposals listed in the table and it is slightly larger than SHA-2. The area requirement is mainly due to the multiple 64-bit adders used inside the design (in total it uses 231 64-bit adders). BMW ranks second regarding area requirements after Skein; the area is again mainly due to multiple adder/subtractor modules in the design. It is worth mentioning that the high regularity of Skein gives the designer the ability to set trade-offs between area and speed, i.e. Skein with 8 or 16 rounds of Threefish present good trade-offs for area/speed for today's applications.

In Fig. 18, area and time delays of each of the SHA-3 designs considered in this work are compared with respect to those of SHA-2.

Hash	Input Size	Output Size	Compression Function Delay	Number of Cycles	Total Delay	Combinational Area	I/O Register Area	Total Area
BMW	512	256	19.20 ns	1	19.20 ns	$670,887.14\,\mu m^2$	$48,746.61\,\mu m^2$	$719,633.75\,\mu m^2$
Luffa	256	256	9.96 ns	1	9.96 ns	$473,318.04\,\mu m^2$	$60,277.99\,\mu m^2$	$533,596.06\,\mu m^2$
Skein	256	256	81.92 ns	1	81.92 ns	$1,605,827.19\,\mu m^2$	$13,788.05\,\mu m^2$	$1,619,615.24\,\mu m^2$
Skein-1c	256	256	3.49 ns	72	251.28 ns	$64,809.98\,\mu m^2$	$27,927.33\,\mu m^2$	$92,737.32\,\mu m^2$
Shabal	256	256	2.42 ns	16	38.72 ns	$38,244.77\,\mu m^2$	$49,041.86\,\mu m^2$	$87,286.64\,\mu m^2$
Blake	512	256	10.4 ns	11	118.8 ns	$191,408.26\,\mu m^2$	$42,204.91\,\mu m^2$	$233,613.18\,\mu m^2$
SHA-2	512	256	105.1 ns	1	105.1 ns	$1,587,346.92\,\mu m^2$	$29,262.01\,\mu m^2$	$1,616,609.00\,\mu m^2$
SHA-2-1c	512	256	2.40 ns	64	153.6 ns	$30,922.68\,\mu m^2$	$27,808.79\,\mu m^2$	$58,731.47\,\mu m^2$

Table 7. ASIC implementation summary of the different compression functions

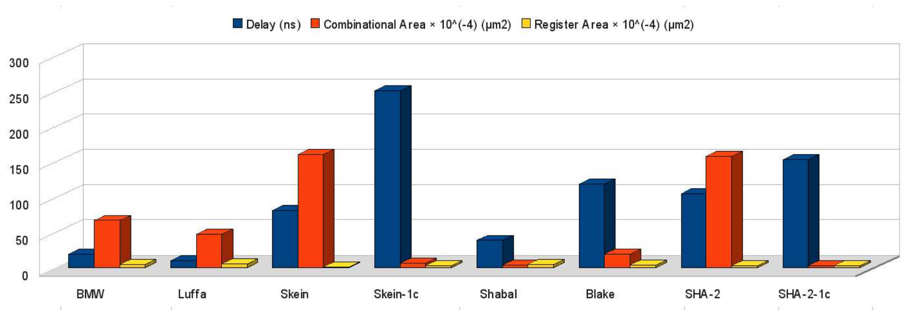

Fig. 18. Area delay complexities for ASIC implementation

9 FPGA Implementation

For our FPGA implementations we have selected Stratix III FPGA family from Altera. The exact device number used is EP3SL340F1760C3 which represents a 1120 I/O pin FPGA using 65 nm technology. Quartus II software package from Altera under UNIX environment has been used to implement the designs.

FPGA implementations have been carried out using the same VHDL files used for ASIC implementations. The only difference was the addition of the serial-in parallel-out (SIPO) and parallel-in serial-out (PISO) modules to transfer the data into or out of the FPGA. Using these modules, the restricted number of I/O pins and the I/O speed limits of the FPGA are taken into account which presents a more practical situation. The details of SIPO and PISO modules are shown in Fig. 19.

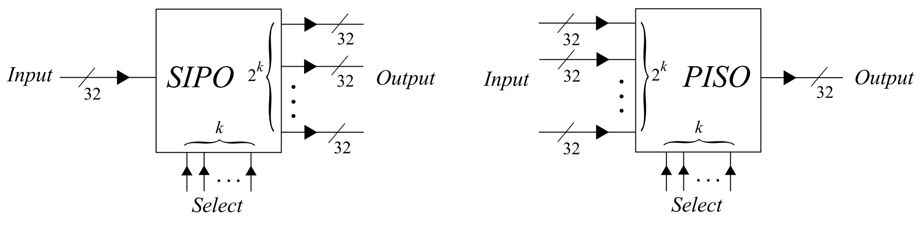

Fig. 19. SIPO and PISO modules used in FPGA implementations

The SIPO module receives the input data as 32-bit words and then stores them for the proper output selected by the input select lines. Using this method input bits can be loaded into the SIPO module 32-bits at a time. Then the data can be transfered to the compression function in parallel. The PISO module uses a similar idea; it loads the data from the compression function in parallel and then transfers them to the output 32-bits at a time.

FPGA implementation results are summarized in Table 8. Quartus was not able to fit the full unrolled version of Skein and SHA-2 in the selected FPGA

Hash	C.F. Clk Frequency	C.F. Clk Cycles	I/O Clk Frequency	I/O Clk Cycles	Total Delay	Combinational ALUTs	Dedicated Logic Registers	I/O Pins
BMW	$9.55\,MHz$	1	$400\,MHz$	32	$184\,ns$	12917	2607	111
Luffa	$47.04\,MHz$	1	$400\,MHz$	16	$61\,ns$	16552	3247	283
Skein	-	-	-	-	-	-	-	-
Skein-1c	$161.42\,MHz$	72	$400\,MHz$	18	$491\,ns$	1385	1858	146
Shabal	$195.35\,MHz$	16	$400\,MHz$	32	$162\,ns$	1440	4000	289
Blake	$46.97\,MHz$	11	$400\,MHz$	24	$294\,ns$	5435	2453	144
SHA-2	-	-	-	-	-	-	-	-
SHA-2-1c	$129.79\,MHz$	64	$400\,MHz$	24	$553\,ns$	813	1582	109

Table 8. FPGA implementation summary of the different compression functions

even though the selected one is the largest available FPGA in Stratix III family. In Table 8, C.F. Clk Frequency presents the maximum frequency that the compression function could be clocked at. C.F. Clk Cycles represents the required number of clock cycles for the compression function to process the input. The I/O Clk Frequency represents the maximum speed at which the SIPO and PISO modules could be clocked to load the data into and out of the FPGA. The I/O speed is limited by the FPGA itself which in our case is $2.5\,ns$ [20]. Total Delay represents the compression function delay in addition to the I/O read and write delay to process a block of data.

The area usage of the FPGA is shown as two separate columns; Combinational ALUTs and Dedicated Logic Registers. Combinational ALUTs represents the number of Adaptive Look-Up Tables (ALUTs) used inside the FPGA. It can be used as a measure of the area used by the combinational logic. Dedicated Logic Registers represents the number of memory cells used inside the FPGA. It can be used as a measure of memory usage.

As can be seen from the table, Luffa presents the fastest architecture ($61\,ns$) followed by Shabal ($162\,ns$) and Blue Midnight Wish ($184\,ns$). Unlike in our ASIC implementations, Blue Midnight Wish is dropped from the second place to the third and Shabal is moved up. This might be the result of having a simple small architecture for Shabal versus extensive use of multiple input 32-bit registers for Blue Midnight Wish. Skein-1c and Blake are ranked as the slowest designs the same way they are ranked in our ASIC implementations. Regarding area usage, Skein-1c requires the smallest area followed by Shabal and Blake.

Note that the area requirements of Skein-1c and Shabal are still larger than SHA-2-1c, while their delays are smaller. In Fig. 20, area and time delays of each of the SHA-3 designs on FPGA are compared with respect to those of SHA-2.

10 Conclusions

In this work we have presented hardware implementation results of the compression function block for five SHA-3 candidates (Luffa, Blue Midnight Wish, Skein, Shabal, and Blake). Hardware implementations have been carried out using the

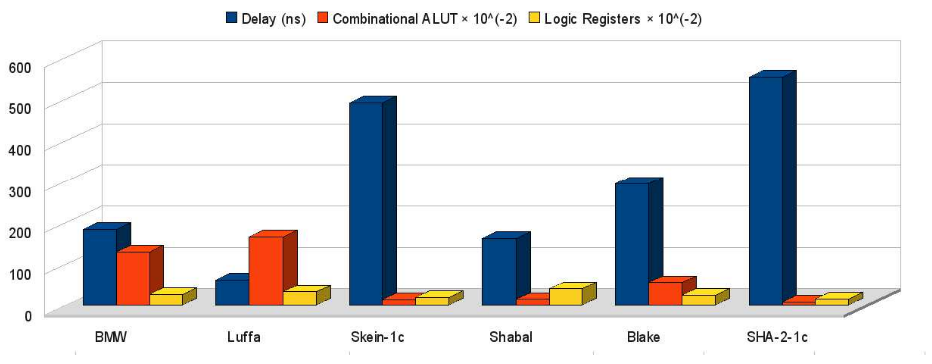

Fig. 20. Area delay complexities for FPGA implementation

90 nm CMOS technology from STMicroelectronics for ASIC and Stratix III from Altera for FPGA. The compression function blocks implemented create the message digest of size 256 bits. Implementation results allow an easy comparison for hardware performance of the selected candidates.

Among our fully unrolled ASIC designs of SHA-3 candidates namely BMW, Luffa, Skein and Blake, Luffa takes the least amount of area and is about three times smaller than SHA-2. For this type of design, Luffa has the least delay and is about ten times faster than SHA-2. For fully unrolled FPGA design, Luffa also ranks at the top for area and delay.

Among our sequential ASIC designs of SHA-3 candidates, namely Skein-1c, Blake and Shabal, the latter outperforms the other two in terms of area as well as delay. For the same sequential type of designs using FPGA, Shabal and Skein-1c rank at the top for delay and area respectively. Each of these three SHA-3 candidates requires more area but less delay than those of sequential SHA-2-1c.

For a given SHA-3 candidate, we are likely to see an improved throughput per unit area in sequential versions of hardware implementation over the fully unrolled version. For example, the throughput per unit area for fully unrolled Skein is $\frac{Input\ Size}{Delay \times Area} = 1,929.4\, bits/(sec\, \mu m^2)$, while that for Skein-1c is $10,985.7\, bits/(sec\, \mu m^2)$. Other hash algorithms, such as Blake and Shabal, are also likely to have such improved utilizations of hardware in their sequential designs.

Acknowledgments

This work was supported in part by NSERC strategic project grant awarded to Dr. Hasan. The authors would also like to thank Olga Nam for her assistance with debugging and testing of the VHDL codes.

References

1. Menezes, A.J., Van Oorschot, P.C., Vanstone, S.A.: Handbook of Applied Cryptography. The CRC Press series on discrete mathematics and its applications, pp. 321-383, 1997.
2. National Institute of Standard and Technology (NIST): Cryptographic Hash Algorithm Competition Website: http://csrc.nist.gov/groups/ST/hash/sha-3/.
3. The SHA-3 Zoo - The ECRYPT hash function website. Website: http://ehash.iaik.tugraz.at/wiki/The_SHA-3_Zoo.
4. European Network of Excellence for Cryptology II (ECRYPT II): ECRYPT Benchmarking of All Submitted Hashes (EBASH): http://bench.cr.yp.to/ebash.html.
5. Fleischmann, E., Forler, C., Gorski, M.: Classification of the SHA-3 Candidates. International Association for Cryptologic Research (IACR) ePrint archive
6. Wang, X., Yao, A., Yao, F.: New Collision search for SHA-1. Crypto 2005 rump sesson.
7. Gligoroski, D., Klima, V., Knapskog S.J., El-Hadedy, M., Amundsen, J., Mjlsnes, S.T.: Cryptographic Hash Function BLUE MIDNIGHT WISH. Submission to NIST, 2008.
8. Canniere, C.D., Sato, H., Watanabe, D.: Hash Function Luffa: Supporting Document. Submission to NIST, 2008.
9. Ferguson, N., Lucks, S., Schneier, B., Whiting, D., Bellare, M., Kohno, T., Callas, J., Walker, J.: The Skein Hash Function Family. Submission to NIST, 2008.
10. Bresson, E., Canteaut, A., Chevallier-Mames, B., Clavier, C., Fuhr, T., Gouget, A., Icart, T., Misarsky, J.F., Naya-Plasencia, M., Paillier, P., Pornin, T., Reinhard, J.R., Thuillet, C., Videau, M.: Shabal, a Submission to NISTs Cryptographic Hash Algorithm Competition. Submission to NIST, 2008.
11. Aumasson,J.P., Henzen, L., Meier, W., C.-W. Phan. R.: SHA-3 proposal BLAKE. Submission to NIST, 2008.
12. Joux, A.: Multicollisions in iterated hash functions. Application to cascaded constructions. In: Proceeding of CRYPTO 2004. LNCS, vol. 3152, pp. 430440, 2004.
13. Lucks, S.: A failure-friendly design principle for hash functions. In: ASIACRYPT, 2005.
14. Liskov, M., Rivest, R., Wagner, D.: Tweakable Block Ciphers. In: Advances in Cryptology-CRYPTO 2002 Proceedings, Springer-Verlag, pp. 3146, 2002.
15. Matyas, S.M., Meyer, C.H., Oseas, J.: Generating strong one-way functions with cryptographic algorithms. IBM Technical Disclosure Bulletin, vol. 27, no. 10A, pp. 5658-5659, 1985.
16. Biham, E., Dunkelman. O.: A framework for iterative hash functions - HAIFA. ePrint report 2007/278, 2007.
17. J. Bernstein, D.J.: ChaCha, a variant of Salsa20. Web: http://cr.yp.to/chacha.html.
18. Aumasson, J.P., Meier, Phan, R. C.-W.: The hash function family LAKE. In: 15th International Workshop on Fast Software Encryption, Lecture Notes in Computer Science, pp. 36-53, 2008.
19. Lucks, S.: A failure-friendly design principle for hash functions. In: ASIACRYPT, 2005.
20. ALTERA: Stratix III Device Handbook, Volume 1, May 2009.
21. Federal Information Processing Standards Publication (FIPS PUB) 180-2: Secure Hash Signature Standard (SHS), pp. 1-79, Aug. 2002.

A Appendix - SHA-2 Hash Function

A.1 Algorithm Description

The basic data block used in SHA-2 (SHA-256) is a word which is 32 bits long. SHA-2 makes use of bit-wise logical word XOR and AND operations along with word addition (modulo 2^{32}), rotate right and shift right operations in its structure. SHA-2's compression function is made of 64 consecutive round functions. The structure of a round function is shown in Fig. 21.

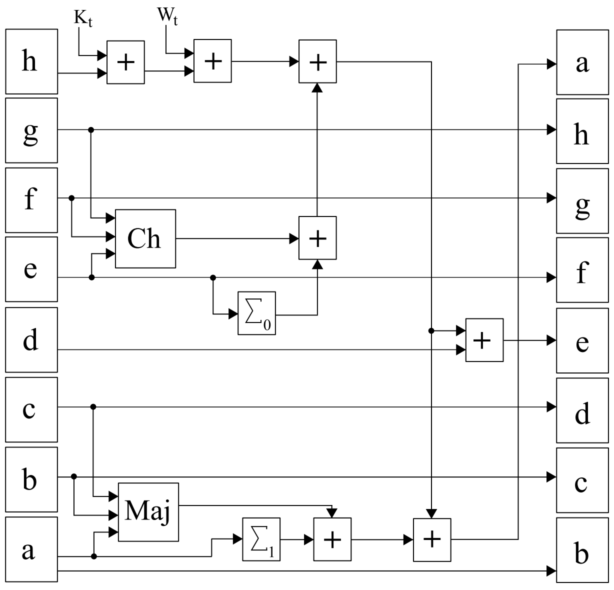

Fig. 21. Block diagram of round function for SHA-2

The inputs to the each round are eight working variables (a, b, c, d, e, f, g, h) 32-bits each along with the two external variables K_t and W_t. The set of working variables $(a, b, c, d.e.f.g.h)$ are initialized at the start of the first round calculation with the previous hash value as follows:

$$\begin{aligned}
a &= H_0^{(i-1)} \\
b &= H_1^{(i-1)} \\
c &= H_2^{(i-1)} \\
d &= H_3^{(i-1)} \\
e &= H_4^{(i-1)} \\
f &= H_5^{(i-1)} \\
g &= H_6^{(i-1)} \\
h &= H_7^{(i-1)}
\end{aligned} \tag{13}$$

K_t variables are 32-bit constants and k_t variables are created through the message schedule process as follows:

$$W_t = \begin{cases} M_t & 0 \leq t \leq 15 \\ \sigma_1(W_{t-2}) + W_{t-7} + \sigma_0(W_{t-15}) + W_{t-16} & 16 \leq t \leq 63. \end{cases} \tag{14}$$

Logical functions σ_0 and σ_1 along with the functions used inside the round function (Ch, Maj,$\sum_0$,$\sum_1$) are defined as follows:

$$\begin{aligned}
\sigma_0 &= ROTR^7(x) \oplus ROTR^{18}(x) \oplus SHR^3(x) \\
\sigma_1 &= ROTR^{17}(x) \oplus ROTR^{19}(x) \oplus SHR^{10}(x) \\
\sum_0 &= ROTR^2(x) \oplus ROTR^{13}(x) \oplus ROTR^{22}(x) \\
\sum_1 &= ROTR^6(x) \oplus ROTR^{11}(x) \oplus ROTR^{25}(x) \\
Ch(x,y,z) &= (x \wedge y) \oplus (\neg x \wedge z) \\
Maj(x,y,z) &= (x \wedge y) \oplus (x \wedge z) \oplus (y \wedge z)
\end{aligned} \tag{15}$$

In (15), ROTR represents a rotate right operation and SHR represents a shift right operations. Also, symbol $\wedge$ represents a logical AND operations, $\oplus$ represents a logical XOR operation and $\neg$ represents a logical complements operation.

After sixty four rounds of calculations, the output hash is created by the addition (modulo 2^{32}) of the last set of working variables (a, b, c, d, e, f, g, h) with the previous hash values as follows:

$$\begin{aligned}
H_0^{(i)} &= a + H_0^{(i-1)} \\
H_1^{(i)} &= b + H_1^{(i-1)} \\
H_2^{(i)} &= c + H_2^{(i-1)} \\
H_3^{(i)} &= d + H_3^{(i-1)} \\
H_4^{(i)} &= e + H_4^{(i-1)} \\
H_5^{(i)} &= f + H_5^{(i-1)} \\
H_6^{(i)} &= g + H_6^{(i-1)} \\
H_7^{(i)} &= h + H_7^{(i-1)}
\end{aligned} \tag{16}$$

A.2 ASIC Implementation

Fully Unrolled/Parallel Design

The architecture used for the ASIC implementation of the SHA-2 is shown in Fig. 22. The previous hash value $H^{(i-1)}$ is stored inside the 256-bit input registers (In_Reg). All the round functions have been realized completely as combinational circuits. After sixty four rounds, one level of adders (eight adders in parallel), exists to create the output hash value according to Eq. (16). Output hash values H^i can be read from the output register(Out_Reg) after one clock cycle.

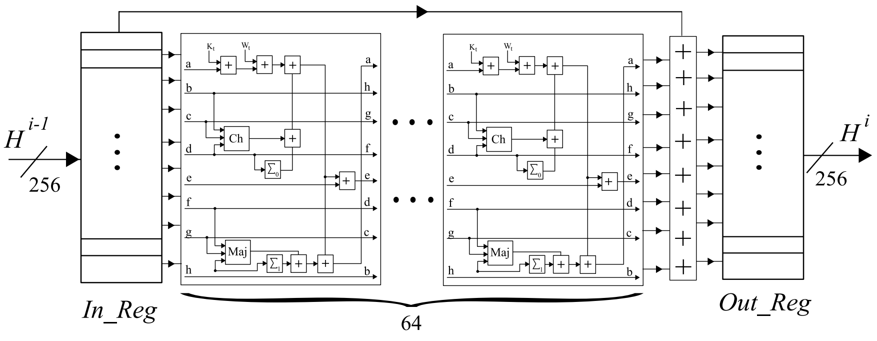

Fig. 22. Hardware architecture used for the compression function of SHA-2

Synthesis results of the SHA-2 compression function are summarized in Table 9. In this table, the Combinational Area represents the area mainly used by the 456 32-bit adders exist in the architecture. Small part of the area utilization is used by the sixty four instances of the $\sum_0, \sum_1, Ch$, and Maj logical functions.

Input Size	Compression Function Delay	Combinational Area	I/O Register Area	Number of Gates
512	$105.1\,ns$	$1,587,346.92\,\mu m^2$	$29,262.01\,\mu m^2$	$168,193$

Table 9. ASIC implementation summary of the SHA-2 compression function

Sequential Design

Similar to Skein, SHA-2 compression function architecture presents a regular design as shown in Fig. 22. This property can be exploited to create a complete SHA-2 compression function by just implementing one out of sixty four rounds inside the compression function. We have implemented a second version of the SHA-2 algorithm (SHA-2-1c) using just one round which is suitable for area constrained applications. The hardware architecture used for this purpose is shown in Fig. 23.

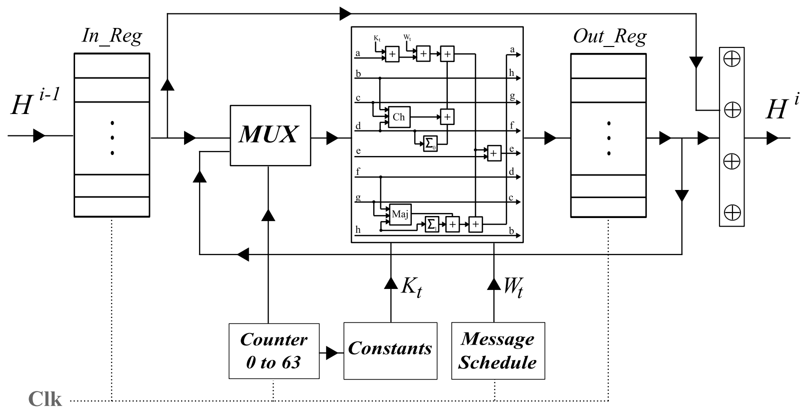

Fig. 23. Hardware architecture used for the compression function of SHA-2-1c

Note that some modifications have been made to the original round function of SHA-2, to make the design capable of creating the message digest. The main modification is the addition of the message schedule module in charge of generating the W_t parameters from the input message blocks according to Eq. (14). This has to be done through a recursive process to present the small delay/area requirements of the design.

Hardware architecture used for this purpose is shown in Fig 24. The message schedule module operates as follows: Initially all input message blocks $(M_0, \ldots, M_15)$ are loaded into the shift register. The shift register itself is made of sixteen registers each made of 32 flip-flops working in parallel. An addition module along with two other modules (σ_0, σ_1) exist which create the W_t parameters for $t \geq 16$ and load them back into the circular shift register.

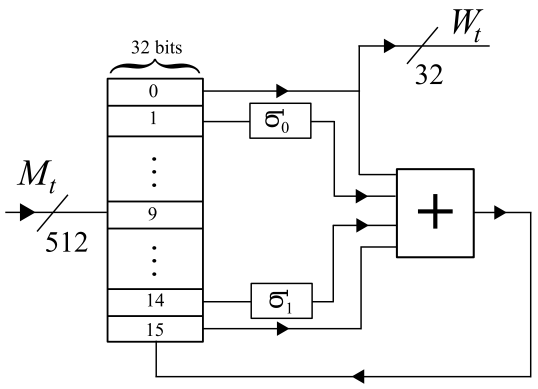

Fig. 24. Hardware architecture of the message schedule module inside the compression function of SHA-2-1c

Also a multiplexer, a counter and a constant module in charge of generating the constants for each round exist inside the design as shown in Fig. 23. Synthesis results for SHA-2-1c compression function are summarized in Table 10.

Input Size	Critical Path Delay	Number of Cycles	Combinational Area	I/O Register Area	Number of Gates
256	$2.40\,ns$	64	$30,922.68\,\mu m^2$	$27,808.79\,\mu m^2$	$4,207$

Table 10. ASIC implementation summary of the SHA2-1c compression function

From Tables 10 and 9 the following can be deduced: the area requirement for the SHA-2 compression function has been reduced from $1,616,609.00\,\mu m^2$ in SHA-2 to $58,731.47\,\mu m^2$ in SHA-2-1c. This comes at the price of increasing the delay from $105.1ns$ in SHA-2- to $153.6\,ns$ in SHA-2-1c.