Research Article

High Performance and Low Power Hardware Implementation for Cryptographic Hash Functions

Yunlong Zhang, 1 Joohee Kim, 1 Ken Choi, 1 and Taeshik Shon 2

1 Electrical and Computer Engineering, Illinois Institute of Technology, Chicago, IL 60616, USA
2 Division of Information and Computer Engineering, College of Information Technology, Ajou University, San 5, Woncheon-Dong, Yeongtong-Gu, Suwon 443-749, Republic of Korea

Correspondence should be addressed to Ken Choi; kchoi@ece.iit.edu

Received 12 September 2013; Accepted 4 January 2014; Published 2 March 2014

Academic Editor: Jongsung Kim

Copyright © 2014 Yunlong Zhang et al. This is an open access article distributed under the Creative Commons Attribution License, which permits unrestricted use, distribution, and reproduction in any medium, provided the original work is properly cited.

Since hash functions are cryptography’s most widely used primitives, efficient hardware implementation of hash functions is of critical importance. The proposed high performance hardware implementation of the hash functions used sponge construction which generates desired length digest, considering two key design metrics: throughput and power consumption. Firstly, this paper introduces unfolding transformation which increases the throughput of hash function and pipelining and parallelism design techniques which reduce the delay. Secondly, we propose a frequency trade-off technique which can give us a scope of frequency value for making a trade-off between low dynamic power consumption and high throughput. Finally, we use load-enable based clock gating scheme to eliminate wasted toggle rate of signals in the idle mode of hash encryption system. We demonstrated the proposed design techniques by using 45 nm CMOS technology at 10 MHz. The results show that we can achieve up to 47.97 times higher throughput, 6.31% delay reduction, and 13.65% dynamic power reduction.

1. Introduction

The explosion of e-commerce nowadays boosts the transaction over the internet; thus we have to prevent intruders from accessing the sensitive information. According to this circumstance, we call for higher security level protection. There are many types of modern cryptography, for example, symmetric-key cryptography, public-key cryptography, and cryptographic hash function. Cryptographic hash function is used in almost every modern application, especially in a multitude of protocols, be it as digital signatures for achieving message authentication and integrity protection. For example, hash-based message authentication codes (HMACs) are used in IP security protocol and also in secure sockets layer (SSL) protocol [1].

As we know, some hash functions, such as message-digest algorithm (MD) series (MD4 and its strengthened variant MD5) and secure hash algorithm (SHA) series (SHA-0 and SHA-1), were widely used, however, broken in practice. Considering the potential danger of being attacked for SHA-2, in 2008, the National Institute of Standards and Technology (NIST) has started the NIST hash competition to develop the future hash standard SHA-3 [2].

Although software encryption is becoming more prevalent today, hardware design is the embodiment of choice for many commercial applications and military [3]. Firstly hardware design is much faster than the corresponding software implementation [4]. Secondly, hardware implementation provides physical protection as high level of security [5]. However, higher security level hash function means more complicated gates, and much more information needs higher frequency to improve the efficiency (or throughput). As a result, the power dissipation of hardware design would increase tremendously. This will cause serious problems in hardware systems, such as less reliability, higher energy consumption, and higher device costs. Thus, low power techniques are highly appreciated in nowadays hardware design.
The rest of this paper is organized as follows. Sponge construction and low power methods which are used in this paper will be introduced in Section 2. In Section 3, we analyze the hash function designed by sponge construction and its original hardware implementation, and then unfolding transformation and pipelining and parallelism design techniques used to improve the throughput and delay of hash function are presented. In Section 4, we construct the hash encryption system and introduce two low power techniques, the frequency trade-off technique and load-enable based clock gating scheme. This paper is concluded in Section 5.

2. Background of the Research

In this section, first, sponge construction will be explained. Next, we will introduce two dynamic power reduction methods which are used in this paper.

2.1. Sponge Construction. The idea of sponge construction came from the design of RadioGatun, and its final definition was given at the Ecrypt Hash Workshop in Barcelona [6]. As shown in Figure 1, sponge construction takes arbitrary length input with finite internal state and gives an output of any desired length.

There are three components in sponge construction [7]:

(i) a state memory;
(ii) a function of fixed length that permutes or transforms the state memory;
(iii) a padding function.

The state memory in Figure 1 is divided into two parts: the top section called bitrate of br bits and the bottom section called capacity of c bits. And the input message (M in Figure 1) will be padded as a whole multiple of the bitrate. Thus this padded input message could be broken into many br-bit blocks.

Sponge construction consists of two processes: absorbing and squeezing. Considering the left part of dash line in Figure 1, called absorbing, firstly, the input message is padded and the state memory will be initialized; secondly, the first br-bit block of padded input will be XORed with the initial br-bit of state memory; thirdly, the fixed length function (block f in Figure 1) updates the state memory. Then steps two and three will be repeated until all the padded br-bit blocks are used up. Considering the right section which is squeezing, firstly, the br-bit of the latest state memory is the first br-bit output; secondly, if we need more output bits, the fixed length function is used to update the state memory and the br-bit of new state memory is the second br-bit output. This process is repeated until the desired number of output bits (Z in Figure 1) is produced.

The extent c-bit part which is altered by the input message depends on the fixed length function [7]. The security of hash function, for example, resistance to collision or preimage attacks, relies on this c-bit part. Because of its arbitrarily long input and output sizes, the sponge construction allows building various primitives such as hash function. Keccak hash function, known as the new SHA-3, uses this sponge construction.

2.2. Dynamic Power Reduction Methods. Digital circuits will consume dynamic power in the active mode. There are two sources of dynamic power consumption [8]:

(i) charging and discharging processes of output capacitance;
(ii) short-circuit current when PMOS and NMOS networks are all ON.

Because the short circuit power is usually less than 10% of total dynamic power [9], the dynamic power consumption which we try to reduce in this paper is referred to as switching power for the rest of this paper. Dynamic power can be explained in (1). Note that f is the clock frequency and TR is the toggle rate of gate output:

\[ P_{\text{dynamic}} = \frac{1}{2} C_L V_{DD}^2 f \cdot \text{TR}. \]  

Since the power optimization at RTL has significant impact with reasonable accuracy, RTL is considered as the optimal stage for low power techniques [8]. According to (1), four parameters, such as voltage, clock frequency, load capacitance, and the toggle rate of gate output, determine the dynamic power consumption. Because reducing supply voltage will increase critical path delay and changing the capacitance of gate output needs to redesign the load logic, it is more efficient to focus on clock frequency and toggle rate at RTL.

2.2.1. Dynamic Voltage/Frequency Scaling. Figure 2 gives us a basic dynamic voltage/frequency scaling (DVFS) system. The DVFS controller will determine the clock frequency, which is sufficient to finish work and gives the best performance without overheating by collecting information about the workload and the temperature. Then this variable clock frequency scheme will lead to dynamic power reduction by choosing proper clock frequency.

2.2.2. Load-Enable Based Clock Gating. As we all know, combinational clock gating technique is widely used to solve dynamic power issue for single level register. And sequential clock gating method considers multiple level (pipeline) registers. In this research, we focus on the combinational clock
International Journal of Distributed Sensor Networks

Figure 2: DVFS system [9].

Figure 3: Load-enable based clock gating.

gating technique; particularly, we use load-enable based clock gating scheme [10].

Figure 3 shows a normal structure of load-enable based clock gating scheme. As we know, if the data do not change during some consecutive clock periods or the enable signal is kept low, those clock periods are wasted. This technique can be applied to a circuit with mux in which an enable signal is a selection signal or a pipeline construction circuit, such as hash encryption system in this research.

3. Proposed High-Speed Hashing Module in Hardware

Cryptographic hash function provides powerful protection for data; it has been utilized in the security layer of every communication protocol. However, as protocols evolve, data sizes and communication speeds are dramatically increasing; low throughput of hash function seems to be a bottleneck in these digital communications systems. A promising solution is the hardware implementation on reconfigurable devices which combines high flexibility with the speed and physical security.

Various techniques have been proposed to speed up or to improve the throughput of hash function, for example, unfolding transformation and pipeline and parallelism techniques. In this section, the characteristics which are relevant to the hardware implementation of the hash algorithm will be presented. Then the high-speed hashing methodology module will be introduced based on the delay bound analysis. Then two techniques, such as unfolding transformation and pipeline and parallelism, will be used to optimize the inner logic of transformation rounds.

3.1. Hash Algorithm Specification. In this section, we introduce a cryptographic hash algorithm with sponge construction, called sponge hash algorithm (SHAT). SHAT is a hash function generating 128-/256-/384-bit hash values. According to the hash value length, SHAT can be denoted by SHAT-(128⋅𝑖) (𝑖 = 1, 2, 3). The parameters of SHAT are shown in Table 1.

3.1.1. 𝐺 Function. 𝐺 function of SHAT consists of an S-box and a diffusion layer. S-box is a substitution function that satisfies the confusion property on each 4-bit word. A 32-bit input word \( W \), for example, is divided into eight 4-bit words \( (w_0, \ldots, w_7) \). Each 4-bit word needs to go through this S-box. The definition of the S-box is \( s_w = Sbox(w) \). This S-box is specified in Table 2. The diffusion layer is a permutation that satisfies the diffusion property (the same as the \( P \) function of Camellia [11]). Considering computational efficiency, this diffusion layer should be represented using only bit-wise exclusive ORs. The branch number of diffusion layer

\[
\begin{pmatrix}
\omega_0 \\
\omega_1 \\
\omega_2 \\
\omega_3 \\
\omega_4 \\
\omega_5 \\
\omega_6 \\
\omega_7
\end{pmatrix}
= \begin{pmatrix}
01111001 \\
10111100 \\
11010110 \\
11000011 \\
01111110 \\
10110111 \\
11011011 \\
11101101
\end{pmatrix}
\begin{pmatrix}
\omega_0 \\
\omega_1 \\
\omega_2 \\
\omega_3 \\
\omega_4 \\
\omega_5 \\
\omega_6 \\
\omega_7
\end{pmatrix}
\] (2)

should be optimal against differential and linear cryptanalysis for security [11]. When we get all eight 4-bit outputs of S-box \( (s_w_0, \ldots, s_w_7) \), this diffusion layer mixes them. Diffusion layer is defined as (2).

3.1.2. Hash Function of SHAT. SHAT uses the hermetic sponge construction as shown in Figure 4. As we mentioned in Section 2, \( br \) is called bitrate and \( c \) is called capacity. And
Table 2: S-box of the G function.

| sw | Sbox(sw) | sw | Sbox(sw) |
|----|----------|----|----------|
| 0×0 | 0×1      | 0×7 | 0×F      |
| 0×1 | 0×2      | 0×8 | 0×F      |
| 0×2 | 0×4      | 0×9 | 0×F      |
| 0×3 | 0×B      | 0×A | 0×F      |
| 0×4 | 0×D      | 0×C | 0×F      |
| 0×5 | 0×E      | 0×D | 0×3      |
| 0×6 | 0×A      | 0×E | 0×0      |
| 0×7 | 0×5      | 0×F | 0×C      |

Table 2: S-box of the G function.

The bit rate (br) and the capacity (c) of SHAT-(128 · i) (i = 1, 2, 3) are 32 · i and 96 · i, respectively. The internal state, S, is divided into 4 · i (i = 1, 2, 3) sections as S = (S₀, . . . , S₄·i−1) (i = 1, 2, 3).

In the absorbing phase, the input message M = (M₀, M₁, . . . , M₄·i−1) shown in Figure 4 is padded as a whole multiple of bitrate (br). Then we will explain our padding method; l is the total length of input message (we assume that l is whole multiple of four as integer multiples of hexadecimal number), and then we append 1 to the end of the message, followed by k bits zero where k is the smallest nonnegative integer to set up the following formulation:

\[(l + 1 + k) \mod (32 \cdot i) = 0.\] (3)

Then, we set S₄·i−1 as the bitrate that used to be XORed with the padded br-bit message block. Then the result goes through that one-way compression function, Perm. Perm is a permutation process which has 48 steps. Each STEP is defined in Algorithm 1. In Algorithm 1, the left circular rotations rot₀ are rot₀ = 19, rot₁ = 1, and rot₂ = 14. In the squeezing phase, SHAT was defined in (4). This SHAT-(128 · i) (i = 1, 2, 3) is specified in Algorithm 2:

\[\text{SQUEEZE}(S, i) = \begin{cases} S₁, & i = 1; \\ S₃ \parallel S₇, & i = 2; \\ S₃ \parallel S₇ \parallel S₁₁, & i = 3. \end{cases}\] (4)

3.2. Hardware Implementation. Following the guidelines of SHAT-(128 · i) (i = 1, 2, 3) as shown in Algorithm 2, the architecture of SHAT is illustrated in Figure 5.

S-box of G function is designed from Karnaugh map. According to Table 2, we get the logic functions of S-box as shown in (5). We set Aᵢ (i = 0, 1, 2, 3) as the input bit of S-box and Qᵢ (i = 0, 1, 2, 3) as the output bit:

\[Q₃ = \overline{A₃} \overline{A₃} A₁ A₀ + A₁ A₂ A₁ A₀ + \overline{A₃} A₂ A₁\]
\[+ \overline{A₃} A₂ A₀ + A₃ A₂ A₁ + A₃ A₂ A₀,\]
\[Q₂ = A₃ A₃ A₁ A₀ + A₃ A₃ A₁ A₀ + A₃ A₁ A₀\]
\[+ \overline{A₃} A₂ A₀ + \overline{A₃} A₂ A₁ A₀,\]
\[Q₁ = A₃ A₃ A₁ A₀ + A₂ A₁ A₀ + \overline{A₃} A₂ A₀\]
\[+ \overline{A₂} A₁ A₀ + A₃ A₂ A₁ A₀,\]
\[Q₀ = \overline{A₃} A₃ A₀ + A₃ A₁ A₀ + A₃ A₂ A₁ A₀\]
\[+ A₃ A₂ A₀ + A₃ A₂ A₁.\] (5)

There are 48 iteration rounds in the basic architecture of Perm function. Then we use rolling loop technique to reduce area requirement. Our design is a single operation block which is reused 48 times as shown in Figure 6. Here rᵢ (i = 1 to 47) is a counter for the number of iteration rounds from 0 to 47. The critical path is highlighted by bold line. Since the delay of circular shift is negligible in hardware implementation, the critical path delay of this architecture is shown as

\[\overline{T}_n = 4 \cdot \text{Delay (⊕)} + \text{Delay (g)}.\] (6)

3.3. Proposed High-Speed Module. In the previous section, we introduce rolling loop technique to construct Perm function. Although this approach considers area efficiency, throughput is kept low due to the requirement of 48 clock cycles to generate the result. There are many architectures that can be made by varying the Perm function to solve this problem. We performed the unfolding transformation technique. This high-speed module combines STEP blocks into a single round and even can take advantage of architectures with complete round-unrolled circuit. By unfolding, the hidden concurrents can be parallelized [12]. Also in [13], the pipeline and parallelism technique was explained to improve the unfolding construction of hash function. This technique is related to precomputing by analysing the inner logic and architecture of hash function.

3.3.1. Unfolding Transformation. According to Figure 6, the mathematical expression of one iteration round is described as

\[S'_₃ = \text{ROT}(S'_₃) \oplus (S'_₃ \oplus S₂),\]
\[S'_₂ = S₁,\]
\[S'_₁ = G(S₃ \oplus r \oplus S₂) \oplus (S₀ \oplus S₁),\]
\[S'_₀ = S₃ \oplus r.\] (7)
Step(S)

(i) For \( k = 0 \) to \( i - 1 \)
   (a) \( S_{4k+3} = S_{4k+3} \oplus r \);
   (b) \( S_{4k} = S_{4k} \oplus S_{4k+1} \);
   (c) \( S_{4k+2} = S_{4k+2} \oplus S_{4k+3} \);
   (d) \( S_{4k} = S_{4k} \oplus G(S_{4k+2}) \);
   (e) \( S_{4k+2} = S_{4k+2} \oplus (S_{4k} \ll \text{rot}_k) \);

(ii) Temp = \( S_{4i-1} \);

(iii) For \( k = 4i - 1 \) to 1
   \( S_k = S_{k-1} \);

(iv) \( S_0 = \) Temp;

Algorithm 1: Typical one step algorithm.

Algorithm 2: SHAT-(128 \cdot i).

Here \( S_i \) (\( i = 0, 1, 2, 3 \)) is the input of current round and \( S'_i \)
(\( i = 0, 1, 2, 3 \)) is the output of this round (or input of next
round). In order to distribute 48 operations equally over each
round, the possible values for unfolding factors are divisors
of 48, that is, 1, 2, 3, 4, 6, 8, 12, 16, 24, and 48. For example,
we can unfold two STEP operations in each round; then we
get 24 rounds in one permutation process. The expression of
throughput is given as

\[
\text{Throughput} = \frac{\text{# of bits}}{\text{# of rounds}} \cdot \frac{f_{\text{round}}}{8},
\]  

(8)

Considering (7), although this unfolding transformation
reduces the maximum operation frequency, the throughput
is increased significantly due to the fact that the operation
numbers are reduced from 48 to 24. The mathematical expression of one iteration round is replaced by
\[ \text{temp}_3 = \text{ROT} (\text{temp}_1) \oplus (\text{temp}_0 \oplus S_2), \]
\[ \text{temp}_2 = S_1, \]
\[ \text{temp}_1 = G (S_1 \oplus r \oplus S_2) \oplus (S_0 \oplus S_1), \]
\[ \text{temp}_0 = S_3 \oplus r, \]
\[ S'_0 = \text{ROT} (S'_1) \oplus (S'_0 \oplus \text{temp}_2), \]
\[ S'_1 = \text{temp}_1, \]
\[ S'_2 = G (\text{temp}_3 \oplus r \oplus \text{temp}_2) \oplus (\text{temp}_0 \oplus \text{temp}_1), \]
\[ S'_3 = \text{temp}_3 \oplus r. \]
(9)

3.3.2. Pipeline and Parallelism. We assume to unroll two STEP operations in each round; for sure it will reduce the frequency to increase the throughput. However, the increased area is introduced as penalty. If some logics can be done in parallel, and this parallelism happens in critical path, then the delay of each round could be decreased, so that the frequency of each operation will be increased. According to (8), when the number of operations is kept as constant (the number of bits is also kept as constant), the throughput will increase with its frequency. This method could be used in any other hardware implementation of hash function.

For example, Figure 7 shows the architecture of unfolding two STEP operations in one round, which has the minimum critical path delay. The critical path is composed of seven XOR gates and two G functions. By unfolding two STEPs in one round, we have a gain of three 32-bit XOR gates and one critical path delay. The critical path is highlighted by bold line.

In Figure 7, cycle counter \( r_{i+1} \) can be calculated with \( \text{temp}_2 \) first, and then XORed with \( \text{temp}_3 \) in second STEP part. Comparing with the first STEP part where \( r_i \) XORed with \( S_3 \) and then XORed with \( S_2 \), we can figure out that there is another additional component which used to make a calculation with \( \text{temp}_0 \) and \( r_{i+1} \). Because of the mandatory output generation necessity, this area penalty cannot be avoided.

Thus, when we increase the number of unfolding STEP operations, for example, three, four, five, \( \ldots \), each round delay will increase by three 32-bit XOR gates and one G function. Therefore, the normalized delay with unfolding factor \( n \) \((n = 1, 2, 3, \ldots)\) is shown as
\[ \bar{T}_n = \frac{4 \cdot \text{Delay}(\oplus) + \text{Delay}(g) + (n-1) \cdot (3 \cdot \text{Delay}(\oplus) + \text{Delay}(g))}{n}. \]
(10)
When we have a limit of \( n \), (10) could be changed into
\[ \lim_{n \to \infty} \bar{T}_n = 3 \cdot \text{Delay}(\oplus) + \text{Delay}(g). \]
(11)
This is the delay bound of SHAT, which means that a delay of one SHAT operation round cannot be less than this bound.

3.4. Experimental Results. We introduce a measurement of hardware efficiency in (12) [14]. This is the improvement of normal figure of merit (FOM). We assume that the power is proportional to the gate count; then we could divide the metric by another GE instead of power dissipation when we want to trade off throughput for power. Note that one gate equivalent (GE) is equal to the area of two-input NAND gate in 45 nm CMOS technology:
\[ \text{FOM} = \frac{\text{Throughput}}{GE^2}. \]
(12)

Table 3 shows the hardware implementation results of some 128-bit hash functions by using 100 kHz clock frequency and 45 nm CMOS technology. Firstly, the throughput of SHAT-128 (66.67 kbps) is less than that of other 5 hash algorithms, such as MD4 (112.28 kbps), MD5 (83.66 kbps), H-Present-128-32-round (200 kbps), and ARMADILLO2-B (250 kbps and 1000 kbps). However the area of SHAT-128 is only 28.42% of that of hash functions in average. This results in having the hardware efficiency of SHAT-128 to be 13.12 times higher in average. Secondly, the area of SHAT-128 (1605 GE) is larger than that of 3 hash algorithms, for example, U-QUARK-544-round (1379 GE), PHOTON-128-996-round (1122 GE), and SPONGENT-128-8-bit-2380-round (1060 GE); however, the throughput of SHAT-128 is 94.27 times higher. Thus the FOM of SHAT-128 is 46.75 times higher in average. Finally, the area of SHAT-128 (1605 GE) is less than that of other 4 hash algorithms, for example, H-Present-128-559-round (2330 GE), U-QUARK-68-round (2392 GE), PHOTON-128-156-round (1708 GE), and SPONGENT-128-70-round (1687 GE). And the throughput of SHAT-128 is also 5.95 times higher than that of 4 hash algorithms in average. This results in having the FOM of SHAT-128 to be 9.66 times higher in average.

In Table 4, firstly, the throughput of SHAT-256 is 51.05% of that of Grostl; however, the area of SHAT-256 is only 21.84% of that of Grostl; this results in having 84.47...
### Table 3: Hardware implementation results of some 128-bit hash functions.

| Hash function  | Block size (bits) | Number of operations | Throughput at 100 kHz (kbps) | Area (GE) | FOM  |
|----------------|-------------------|----------------------|-----------------------------|-----------|------|
| SHAT-128       | 32                | 48                   | 66.67                       | 1605      | 258.80 |
|                | 128               | 559                  | 11.45                       | 2330      | 21.09 |
|                | 128               | 32                   | 200                         | 4256      | 110.41 |
| MD4 [15]       | 512               | 456                  | 112.28                      | 7350      | 20.78 |
| MDS [15]       | 512               | 612                  | 83.66                       | 8400      | 11.86 |
| ARAMADILLO2-B [15] | 64               | 256                  | 250                         | 4353      | 13.19 |
|                | 64                | 64                   | 1000                        | 6025      | 27.55 |
| U-QUARK [15]   | 8                 | 544                  | 1.47                        | 1379      | 7.73  |
|                | 8                 | 68                   | 11.76                       | 2392      | 20.56 |
| PHOTON-128 [15] | 16               | 156                  | 10.26                       | 1708      | 35.15 |
|                | 16                | 996                  | 1.61                        | 1122      | 12.78 |
|                | 8                 | 2380                 | 0.34                        | 1060      | 2.99  |
| SPONGENT-128 [15] | 16               | 70                   | 11.43                       | 1687      | 40.16 |

### Table 4: Hardware implementation results of some 256-bit hash functions.

| Hash function  | Block size (bits) | Number of operations | Throughput at 100 kHz (kbps) | Area (GE) | FOM  |
|----------------|-------------------|----------------------|-----------------------------|-----------|------|
| SHAT-256       | 64                | 48                   | 133.33                      | 3193      | 130.78 |
| SHA-256 [14]   | 512               | 490                  | 104.48                      | 8588      | 14.17 |
| ARMADILLO2-E [14] | 128             | 512                  | 25                          | 8653      | 3.34  |
| BLAKE [14]     | 32                | 816                  | 72.79                       | 13575     | 0.21  |
| Grosstl [14]   | 64                | 196                  | 261.14                      | 14622     | 1.53  |
| PHOTON-256 [14] | 32               | 156                  | 3.21                        | 2177      | 6.78  |
|                | 32                | 156                  | 20.51                       | 4362      | 10.17 |
| SPONGENT-256 [14] | 16              | 9520                 | 0.17                        | 1950      | 0.44  |
|                | 16                | 140                  | 11.43                       | 3281      | 10.62 |

### Table 5: Hardware implementation results of some 384-bit hash functions.

| Hash function  | Block size (bits) | Number of operations | Throughput at 100 kHz (kbps) | Area (GE) | FOM  |
|----------------|-------------------|----------------------|-----------------------------|-----------|------|
| SHAT-384       | 96                | 48                   | 200                         | 4753      | 88.53 |
| SHA-384 [14]   | 1024              | 84                   | 1219.04                     | 43330     | 6.49  |

### Table 6: Performance results of hash function using pipeline and parallelism.

| Number of iteration rounds | Area (GE) | Increase (%) | Delay (ns) | Reduction (%) | Power (𝜇W) | Increase (%) |
|---------------------------|-----------|--------------|------------|---------------|------------|--------------|
| 48                        | 965       | 0.00         | 0.94       | 0.00          | 27.27      | 0.00         |
| 24                        | 2010      | 4.15         | 1.87       | 2.60          | 79.05      | 0.91         |
| 16                        | 3055      | 5.53         | 2.81       | 3.44          | 136.42     | 3.52         |
| 12                        | 4100      | 6.22         | 3.74       | 4.35          | 193.71     | 4.67         |
| 8                         | 6190      | 6.91         | 5.61       | 4.92          | 308.48     | 5.73         |
| 6                         | 8280      | 7.25         | 7.47       | 5.32          | 423.18     | 6.21         |
| 4                         | 12460     | 7.60         | 11.20      | 5.64          | 652.62     | 6.68         |
| 3                         | 16640     | 7.77         | 14.93      | 5.74          | 882.00     | 6.90         |
| 2                         | 25000     | 7.94         | 22.40      | 5.88          | 1340.80    | 7.12         |
| 1                         | 50080     | 8.12         | 44.70      | 6.31          | 2695.40    | 7.40         |
Table 7: Performance results of unrolling steps constructions.

| Number of iteration rounds | Area (GE) | Delay (ns) | Power (μW) | Throughput at 10 MHz (Mbps) |
|----------------------------|-----------|------------|------------|-----------------------------|
| 48                         | 965       | 0.94       | 27.27      | 6.67                        |
| 24                         | 1930      | 1.92       | 78.34      | 13.33                       |
| 16                         | 2895      | 2.91       | 131.78     | 20.00                       |
| 12                         | 3860      | 3.91       | 185.06     | 26.67                       |
| 8                          | 5790      | 5.90       | 291.77     | 40.00                       |
| 6                          | 7720      | 7.89       | 398.42     | 53.33                       |
| 4                          | 11580     | 11.87      | 611.78     | 80.00                       |
| 3                          | 15440     | 15.84      | 825.06     | 106.67                      |
| 2                          | 23160     | 23.80      | 1251.70    | 160.00                      |
| 1                          | 46320     | 47.71      | 2509.70    | 320.00                      |

4. Low Power Design for Hash Function

Low power design is a significant consideration in hardware implementation. How much the power consumption is will determine a device’s life, reliability, and energy cost. Thus low power technique is applied normally to every application nowadays. There are many methods to reduce power consumption such as clock gating and power gating related to dynamic power and leakage power. Frequency decreasing technique will pull down the power dissipation dramatically as well.

Firstly, we will propose the frequency trade-off technique. By using this method we could achieve a range of frequency values for making a trade-off between low power consumption and high throughput of hash function. Secondly, we construct a hash encryption system which includes input data padding unit, RAM registers, main hash computing construction, message digest extraction component, and main control unit. Thirdly, by analyzing the idle mode and control signals of this hash encryption system, load-enable based clock gating scheme is applied to reduce the dynamic power consumption.

4.1. Frequency Trade-Off Technique. According to (1), reducing clock frequency is an effective method to decrease dynamic power dissipation linearly. In Section 2.2, we talked about the DVFS technique. By collecting the information about workload and temperature, DVFS will determine the sufficient clock frequency for the proper performance. However, modifying the clock frequency at RTL is not easy. Normally, we treat the clock frequency as constant. Also as we know, dynamic frequency scaling reduces the number of operations a system can issue in a given amount of time, thus reducing performance. Therefore, there is an issue we need to consider: high clock frequency brings high level throughput; however dramatically increased dynamic power consumption is the critical drawback. Low clock frequency minimizes the dynamic power dissipation; however it decreases the throughput as well.

However, according to the unfolding transformation technique which is introduced in Section 3.3, the maximum frequency of Perm function will decrease, while the number of unrolling loops increases. It means that we can decrease the clock frequency while increasing throughput of the hash algorithm. Thus, this unrolling transformation technique compromises high performance without high clock frequency. According to this advantage, by choosing proper clock frequency, we can make a trade-off between high performance and low power consumption.

Next, we explain how to get this scope of frequency value from the two performance bounds. For example, first we achieve two values of rolling Perm circuit: dynamic power consumption $P_1$ and clock frequency $f_1$ which is defined by the necessity of circuit design (the clock period computed from $f_1$ needs to be not less than the critical path delay). Then, according to (8), we can get the throughput $T_1$ at this frequency. Thus, those two performance bounds are defined in (13), where $n$ is the number of iteration rounds in one Perm function with rolling STEPs:

$$P_{\text{max}} = P_1 \cdot n,$$

$$T_{\text{min}} = T_1.$$  

This method can be defined as the following: referring to the performance of original folding circuit (we assume that this circuit is the one with 48 iteration rounds in one
Perm function), each unfolding transformation design with different numbers of unrolling STEPs (2, 3, …, 48) has two performance bounds: one is maximum dynamic power and the other is minimum throughput of the circuit. These two performance bounds are used to determine the boundary of proper frequency range for each unfolding transformation circuit. It means that when we choose one specific clock frequency in this value scope, the total dynamic power consumption of that PERM function will be not more than defined maximum dynamic power \( P_{\text{max}} \) and its throughput will be not less than that fixed minimum throughput \( T_{\text{min}} \).

This clock frequency scope gives us many different choices for different circuit designs by using unfolding transformation technique. The results of this frequency trade-off technique are shown in Table 9 in Section 4.4.

4.2. Hash Encryption System Design. The hash encryption system is divided into 5 main parts as shown in Figure 8.

Firstly, the receiver and RAM section is actually our padding unit. We use serial communication technique to connect PC and the hash encryption system. Thus, we need clock divider to generate proper clock cycle to be synchronous with Baud rate of serial communication. We choose 4800 Baud/s as our transmission Baud rate which is not a quick speed for low error rate (less than 3%). In this case, one Baud represents 1 bit. Our rule of transmission is a one start bit “0”, then 8-bit message, and one finish bit “1”. This start bit and finish bit will be added into the transmission message bits automatically; the sampling rate of receiver is 16 MHz and FPGA board provides 100 MHz clock frequency. Thus, the clock period used in sampling is 1302 times provided 100 MHz clock period as shown in (14). This error is 0.0064% less than 3%:

\[
\text{Sampling Clock Cycles} = \frac{\text{Sampling Rate}}{\text{Baud rate}} = \frac{100 \text{ MHz}}{16 \times 4800 \text{ B/s}} = 1302.
\]

Because the liquid crystal display (LCD) limits the number of characters we can display which are 32 characters in hexadecimal, this number is suitable for the number of digest bits of SHAT-128. Thus, our \( br \) for each padded block is determined to be 32 bits which consist of eight 4-bit hexadecimal numbers.

Secondly, hash function which we introduced in Section 3 is designed as sponge construction as shown in Figure 4. Absorbing \( n \) 32-bit message blocks, there are 128 bits digest that will be squeezed out.

Finally, the main control unit is designed for managing the working order between receiver, hash process, and LCD display. Figure 9 shows the pipeline working of system.

Because we use serial communication technique, the speed will be slow. We apply 4800 Baud/s as our Baud rate for low error rate; thus each 32-bit block needs roughly 7 ms. For example, there are seven 32-bit blocks that need to be transmitted; roughly 50 ms needs to be dissipated for data receiving and padding. Although the hash function that we used in this system is one STEP each round, this means that there are 48 iteration rounds for a complete Perm function. However, hash processing just needs roughly 6 \( \mu \text{s} \). It also costs much time in LCD displaying period. Even though we can finish LCD initialization before we get hash digest, we still need roughly 1.5 ms to completely display all data.

4.3. Load-Enable Based Clock Gating. In this section, we introduce the load-enable based clock gating technique for the hash encryption system.

Clock gating is the most widely used low power technique at RTL. It is more reasonable to determine the toggle rate of gate output at RTL than any other three components, such as \( V_{\text{DD}}, \) clock frequency, and gate output capacitance. According to Figure 9, the hash encryption system is composed of a pipeline construction. Finishing signal of each process can be treated as enable signal in load-enable based clock gating as shown in Figure 3. On the other hand, XOR-based clock gating technique needs to specify the outputs of single level flip-flops which is not easily determined in our encryption system; thus the load-enable based clock gating is our best option for low power method.

As shown in Figure 10, there are three signal pairs to realize this load-enable based clock gating: \( \text{en}_{\text{div}} \) and \( \text{fsh}_{r} \), \( \text{en}_{\text{h}} \) and \( \text{fsh}_{h} \), and \( \text{en}_{\text{jcd}} \) and \( \text{fsh}_{\text{jcd}} \). Because receiver is implemented in a specific clock frequency which is corresponding to the serial communication, the main control unit will not gate the clock signal of receiver directly; by controlling the clock signal of clock divider with \( \text{en}_{\text{div}} \), receiver can be properly managed.

Figure 9 gives us three operation phases of the encryption system. In first phrase, \( \text{en}_{\text{div}} \) and \( \text{en}_{\text{jcd}} \) signals are asserted to logic one and \( \text{en}_{\text{h}} \) is asserted to logic zero; thus receiver starts receiving input messages and padding them into RAM. At the meantime, system will begin the initialization process for LCD displayer. However, the hash processing unit is waiting for the padded input message. Considering the serial communication takes long time due to the low Baud rate and its characteristic which is transmitting message bit one by
Phase one Phase two Phase three
Data receiving and padding Idle Hash processing Idle LCD displaying

Figure 9: Three phases of hash encryption system.

One, LCD display initialization can be finished before the padded message is ready. Thus, \( \text{en}_{\text{lcd}} \) can be asserted to logic zero by main control unit when \( \text{fsh}_{\text{lcd}} \) is switching to logic one.

During the second phase, because the padded message is ready, then \( \text{fsh}_{\text{r}} \) switches to logic one. Then \( \text{en}_{\text{div}} \) is asserted to zero which means that clock divider is turned off; then no specific clock frequency is produced; thus the receiver will stop working. In this phase, \( \text{en}_{\text{h}} \) is asserted to logic one for hash encryption which is our core function. \( \text{en}_{\text{lcd}} \) is still zero waiting for the hash digest generated by hash processing.

This system will enter the third phase when the \( \text{fsh}_{\text{h}} \) signal switches to logic one. In this phase, hash digest is ready; thus both receiver and hash processes are in idle mode which means that \( \text{en}_{\text{div}} \) and \( \text{en}_{\text{h}} \) are all asserted to logic zero. Signal \( \text{en}_{\text{lcd}} \) will be asserted to logic one to start LCD displaying. \( \text{en}_{\text{lcd}} \) will be asserted back to zero when the displaying process is finished. This is the end of the whole system; then the device will be turned off or repeats these three phases for another input message.

By analyzing the construction and process of hash encryption system, we can figure out the idle time for each component. Then applying the load-enable based clock gating to each component, the dynamic power dissipation of this system can be properly reduced as shown in Table 8 in Section 4.4.

### 4.4. Experimental Results

By using 10 MHz clock frequency and 45 nm CMOS technology, the results of frequency trade-off technique are shown in Tables 9, 10, and 11. Table 9 shows that the area and critical path delay are not changed comparing with the unfolding transformation technique. Tables 10 and 11 give us the variation of dynamic power consumption and throughput with frequency trade-off method. Note that \( f_i \) stands for frequency, \( T_i \) stands for throughput, and \( T_{\text{spec}} \) is the percentage of increasing comparing with the minimum throughput \( (T_{\text{min}}) \) which is 6.67 Mbps. \( P_i \) means the total dynamic power consumption by finishing a complete Perm function and \( P_{\text{spec}} \) is the percentage of power reduction comparing with the maximum power consumption \( (P_{\text{max}}) \) defined as 1308.96 \( \mu \)W which is calculated from the product of 48 (number of iteration rounds) and 27.27 \( \mu \)W (as shown in Table 7). Note that \( i \) stands for the number of iteration rounds.

Then we apply load-enable based clock gating scheme to hash encryption system by using 100 MHz clock frequency, which can be provided on FPGA board, and 45 nm CMOS technology. As shown in Table 8, the dynamic power decreases 13.65%. However, 3.64% increased area and 5.52% increased critical path delay are sacrificed.

### Table 8: Hardware implementation with/without load-enable based clock gating.

| System type | Area (GE) | Delay (ns) | Power (\( \mu \)W) | Reduction (%) |
|-------------|-----------|------------|-------------------|---------------|
| Original    | 14053     | n/a        | 1.63              | n/a           |
| Clock gated | 14565     | 3.64       | 1.72              | 1580.36       | 13.65        |

### Table 9: Area and delay performances of frequency trade-off technique.

| Number of iteration rounds | Area (GE) | Delay (ns) | Frequency (MHz) |
|----------------------------|-----------|------------|-----------------|
| 48                         | 965       | 0.94       | 10.00           |
| 24                         | 1930      | 1.92       | 5.00 < \( f_{24} \) < 6.96 |
| 16                         | 2895      | 2.91       | 3.33 < \( f_{16} \) < 6.20 |
| 12                         | 3860      | 3.91       | 2.50 < \( f_{12} \) < 5.89 |
| 8                          | 5790      | 5.90       | 1.67 < \( f_{8} \) < 5.60 |
| 6                          | 7720      | 7.89       | 1.25 < \( f_{6} \) < 5.47 |
| 4                          | 11580     | 11.87      | 0.83 < \( f_{4} \) < 5.34 |
| 3                          | 15440     | 15.84      | 0.63 < \( f_{3} \) < 5.28 |
| 2                          | 23160     | 23.80      | 0.42 < \( f_{2} \) < 5.22 |
| 1                          | 46320     | 47.71      | 0.21 < \( f_{1} \) < 5.21 |
Table 10: Dynamic power consumption of frequency trade-off technique.

| Number of iteration rounds | Power (µW) | Reduction (%) | Frequency (MHz) |
|----------------------------|------------|---------------|-----------------|
| 48                         | 1308.96    | n/a           | 10.00           |
| 24                         | 940.08 < P_{24} < 1308.48 | 28.18 < P_{24ptc} < 0.04 | 5.00 < f_{24} < 6.96 |
| 16                         | 702.88 < P_{16} < 1307.20 | 46.30 < P_{16ptc} < 0.13 | 3.33 < f_{16} < 6.20 |
| 12                         | 555.12 < P_{12} < 1308.00 | 57.59 < P_{12ptc} < 0.07 | 2.50 < f_{12} < 5.89 |
| 8                          | 389.04 < P_{8} < 1307.12 | 70.28 < P_{8ptc} < 0.14 | 1.67 < f_{8} < 5.60 |
| 6                          | 298.80 < P_{6} < 1307.64 | 77.17 < P_{6ptc} < 0.10 | 1.25 < f_{6} < 5.47 |
| 4                          | 203.92 < P_{4} < 1306.76 | 84.42 < P_{4ptc} < 0.17 | 0.83 < f_{4} < 5.34 |
| 3                          | 154.71 < P_{3} < 1306.89 | 88.18 < P_{3ptc} < 0.16 | 0.63 < f_{3} < 5.28 |
| 2                          | 104.30 < P_{2} < 1306.76 | 92.03 < P_{2ptc} < 0.17 | 0.42 < f_{2} < 5.22 |
| 1                          | 52.29 < P_{1} < 1307.60 | 96.01 < P_{1ptc} < 0.10 | 0.21 < f_{1} < 5.21 |

Table 11: Throughput performances of frequency trade-off technique.

| Number of iteration rounds | Throughput (Mbps) | Improvement (%) | Frequency (MHz) |
|----------------------------|------------------|-----------------|-----------------|
| 48                         | 6.67             | n/a             | 10.00           |
| 24                         | 6.67 < T_{24} < 9.28 | 0.00 < T_{24ptc} < 39.13 | 5.00 < f_{24} < 6.96 |
| 16                         | 6.67 < T_{16} < 12.4 | 0.00 < T_{16ptc} < 85.91 | 3.33 < f_{16} < 6.20 |
| 12                         | 6.67 < T_{12} < 15.71 | 0.00 < T_{12ptc} < 135.53 | 2.50 < f_{12} < 5.89 |
| 8                          | 6.67 < T_{8} < 22.40 | 0.00 < T_{8ptc} < 235.83 | 1.67 < f_{8} < 5.60 |
| 6                          | 6.67 < T_{6} < 29.17 | 0.00 < T_{6ptc} < 337.33 | 1.25 < f_{6} < 5.47 |
| 4                          | 6.67 < T_{4} < 42.72 | 0.00 < T_{4ptc} < 540.48 | 0.83 < f_{4} < 5.34 |
| 3                          | 6.67 < T_{3} < 56.32 | 0.00 < T_{3ptc} < 744.38 | 0.63 < f_{3} < 5.28 |
| 2                          | 6.67 < T_{2} < 83.52 | 0.00 < T_{2ptc} < 1152.17 | 0.42 < f_{2} < 5.22 |
| 1                          | 6.67 < T_{1} < 166.72 | 0.00 < T_{1ptc} < 2399.55 | 0.21 < f_{1} < 5.21 |

5. Conclusion

In order to achieve high performance and low power hardware implementation for cryptographic hash function which uses sponge construction, firstly, we use unfolding transformation technique to improve the throughput of hash function; secondly, pipeline and parallelism design techniques are implemented to reduce the critical path delay by modifying the structure of permutation function; thirdly, frequency trade-off technique is proposed to calculate a frequency scope which can be used to make a trade-off between low dynamic power consumption and high throughput of hash function; finally, load-enable based clock gating scheme is applied in hash encryption system to eliminate wasted toggle rate of signals in the idle mode.

The experimental results have shown that unfolding transformation technique can achieve up to 47.97 times higher throughput, pipeline and parallelism methods give 6.31% delay reduction, load-enable based clock gating scheme decreases 13.65% dynamic power consumption, and frequency trade-off technique shows how to decide the clock frequency of the hash function to achieve low power consumption and high throughput.

Conflict of Interests

The authors declare that there is no conflict of interests regarding the publication of this paper.

Acknowledgment

This research was supported by the MKE (The Ministry of Knowledge Economy), Korea, under the ITRC (Information Technology Research Center) support program (NIPA-2012-H0301-12-3007) supervised by the NIPA (National IT Industry Promotion Agency).

References

[1] H. Michail and C. Goutis, "Holistic methodology for designing ultra high-speed SHA-1 hashing cryptographic module in hardware," in Proceedings of the IEEE International Conference on Electron Devices and Solid-State Circuits (EDSSC '08), pp. 1–4, Hong Kong, December 2008.

[2] "Cryptographic hash algorithm competition," NIST Computer Security Resource Center, http://csrc.nist.gov/groups/ST/hash/sha-3/index.html.
[3] B. Schneier, *Applied Cryptography: Protocols, Algorithms, and Source Code in C*, John Wiley & Sons, New York, NY, USA, 2nd edition, 1996.

[4] J. Nakajima and M. Mitsuru, "Performance analysis and parallel implementation of dedicated hash function," in *Proceedings of the International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT’02)*, vol. 2332, pp. 165–180, Amsterdam, The Netherlands, 2002.

[5] P. C. van Oorschot, A. Somayaji, and G. Wurster, "Hardware-assisted circumvention of self-hashing software tamper resistance," *IEEE Transactions on Dependable and Secure Computing*, vol. 2, no. 2, pp. 82–92, 2005.

[6] G. Bertoni, J. Daemen, M. Peeters, and G. van Assche, "Cryptographic sponge functions," The Sponge Functions Corner, http://sponge.noekeon.org/index.html.

[7] "Sponge function," WIKIPEDIA, http://en.wikipedia.org/wiki/Sponge_function.

[8] L. Li, *Power optimization from register transfer level to transistor level in deeply scaled CMOS technology [Ph.D. thesis]*, Illinois Institute of Technology, Chicago, Ill, USA, 2012.

[9] N. Weste and D. Harris, *CMOS VLSI Design: A Circuits and Systems Perspective*, Addison-Wesley, Reading, Mass, USA, 2010.

[10] Y. Zhang, Q. Tong, L. Li et al., "Automatic register transfer level CAD tool design for advanced clock gating and low power schemes," in *Proceeding of the International SoC Design Conference (ISOCC ’12)*, pp. 21–24, Jeju Island, Republic of Korea, 2012.

[11] K. Aoki, T. Ichikawa, and M. Kanda, "Specification of Camellia—a 128-bit block cipher," Nippon Telegraphy and Telephone Corporation, Mitsubishi Electric Corporation, 2000.

[12] Y. K. Lee, H. Chan, and I. Verbauwhede, "Throughput optimized SHA-1 architecture using unfolding transformation," in *Proceedings of the 17th IEEE International Conference on Application-Specific Systems, Architectures and Processors (ASAP ’06)*, pp. 354–359, Steamboat Springs, Colo, USA, September 2006.

[13] H. Michail, A. P. Kakarountas, O. Koufopavlou, and C. E. Goutis, "A low-power and high-throughput implementation of the SHA-1 hash function," in *Proceedings of the IEEE International Symposium on Circuits and Systems (ISCAS ’05)*, vol. 4, pp. 4086–4089, Kobe, Japan, May 2005.

[14] S. Badel, N. Dadjekin, J. Nakahara Jr. et al., "ARMADILLO: a multi-purpose cryptographic primitive dedicated to hardware," in *Cryptographic Hardware and Embedded Systems, CHES 2010*, vol. 6225 of *Lecture Notes in Computer Science*, pp. 398–412, 2010.

[15] K. Lin, Y. Zhang, K. Choi, J. Kang, and S. Hong, "Multi-purpose bimodal cryptographic algorithm and its hardware implementation," in *Proceedings of the FTRA International Conference on Advanced IT, Engineering and Management (FTRA AIM ’13)*, Seoul, Korea, 2013.
