High-Throughput of SHA-256 Hash Function with Unfolding Transformation

Shamsiah binti Suhaili 1*, Takahiro Watanabe2

1 Faculty of Engineering, Universiti Malaysia Sarawak, 94300 Kota Samarahan, Sarawak, Malaysia
2 Graduate School of Information, production and Systems, Waseda University, 2-7 Hibikino, Wakamatsu-ku, Kitakyushu-shi, Fukuoka 808-0135, Japan

ABSTRACT

Hash Function in cryptography algorithms is used to encrypt the message by giving the appropriate output based on the structure of the hash function itself. This algorithm is important for security application such as Keyed-Hash Message Authentication Code (HMAC), digital signature and others. There are different types of hash function such as MD5, SHA-1, RIPEMD-160, SHA-256, SHA-224, SHA-384, SHA-512 and others. In this paper, the unfolding transformation method was proposed to improve the throughput of the SHA-256 hash function. Three types of SHA-256 hash function were designed namely SHA-256 design, SHA-256 design inner pipelining with unfolding factor 2 and SHA-256 design inner pipelining with unfolding factor 4. The designs were written in Verilog code and the output simulations were verified using ModelSim. The simulation results showed that the proposed SHA-256 inner pipelining unfolding with factor 4 provided the highest throughput which is 4196.30 Mbps, and with factor 2 was superior in terms of maximum frequency and was better than the conventional SHA-256 design.

Type of Paper: other.

Keywords: Cryptography algorithm; FPGA; SHA-256 Hash Function; Unfolding transformation, Verilog

1. Introduction

Cryptography is the science of writing secret codes; to ensure none can read an encrypted message except the intended user. There are three different types of cryptographic algorithms namely symmetric cryptography, asymmetric cryptography and hash function. While symmetric cryptography uses only a key to encrypt and decrypt the message, asymmetric cryptography uses two different keys and the hash function requires no key. This paper focused on the SHA-256 hash function. It transformed a variable message input into a fixed size string hash value [1].
The hash function output is called hash code, also referred to as hash value or messages digest. The hash function input can be of any length while the output has a fixed length based on the structure of the hash function. It is a one-way function where it is not feasible to find an input message. In this paper, the unfolding transformation method was proposed to improve the throughput of the SHA-256 hash function. Nowadays, security on the network is a major issue in data transmission. A network layer needs to be secure enough with cryptographic algorithms so that it can be used to accommodate encryption and authentication processes. Therefore, high performance of cryptographic hash function algorithm is one of the important aspects of security algorithm. Hence, designing a cryptographic hash function algorithm on reconfigurable hardware with high speed, low power and small area implementation needs to be considered.

The unfolding transformation method is used to improve the throughput of the SHA-256 hash function, which is the purpose of this study. This method focused on the latency of the designs and the architecture refers to Register Transfer Level (RTL). In this paper, unfolding transformation factor 2 and 4 were used to reduce the latencies of the SHA-256 hash function. There was parallel execution for both designs. The combination of inner pipelining and unfolding resulted in further power reduction because the power supply voltage was reduced aggressively, and the frequency of operation was also reduced with small area implementation.

2. Methodology – Unfolding Design

The motivation of using unfolding method was to improve the performance in terms of throughput. Verilog code was used to design SHA-256. The SHA-256 architecture consists of 6 top-level modules such as counter SHA-256, message schedule, constant SHA-256, multiplexer, compression function and output SHA-256. The difference between the conventional SHA-256 design and SHA-256 unfolding designs are the input of the inner structure of SHA-256 unfolding design. By using different inputs, the sequence of constant and message was found to be also different. The input data was 15 blocks input of padded 32-bit data. Equation (1) was used to obtain the input message, \( W_t \) for \( 16 \leq t \leq 63 \).

Message schedule SHA-256, \( W_t \)

\[
W_t = \text{message input} \\
W_t = \sigma_0^{256}(W_{t-2}) + W_{t-7} + \sigma_1^{256}(W_{t-15}) + W_{t-16} \quad 0 \leq t \leq 15 \\
W_t = \sigma_0^{256}(W_{t-2}) + W_{t-7} + \sigma_1^{256}(W_{t-15}) + W_{t-16} \quad 16 \leq t \leq 63
\]

(1)

Where,

\[
\sigma_0^{256}(x) = ROTR^7(x) + ROTR^{18}(x) + SHR^3(x) \quad (2) \\
\sigma_1^{256}(x) = ROTR^{17}(x) + ROTR^{19}(x) + SHR^0(x) \quad (3)
\]

\( \sigma_0 \) and \( \sigma_1 \) represented sigma 0 and sigma 1. Both functions were obtained from Equation (2) and Equation (3). Equation (2) meant that the message \( x \) was rotated right by 7 bits, the result was then added with a right rotation of \( x \) by 18 bits and finally, the result was added with three shifts to the right. As for Equation (3), the message \( x \) was rotated by 17 bits, the result was then rotated 19 bits to the right and finally shifted right by 10 bits.

A counter module was used to generate the sequence the message. The final hash code was obtained after the 64 rounds of iteration of the compression function used by the SHA-256 hash function. A Multiplexer module helped to generate eight buffer initialisation of SHA-256, before SHA-256 started processing the message. The constant \( Kt \) was defined using 64X32-bit ROM blocks. Finally, the SHA-256 message digest was produced using the output module. The last output of the
compression function of SHA-256 was added with buffer initialisation in this model. Modification must be done for each module in order to improve the performance of SHA-256 in terms of throughput. Two parallel inputs of 32 bits and two parallel constants were needed in order to obtain unfolding factor 2. Similarly, for unfolding factor 4, four parallel inputs of 32 bits and four parallel constants were needed to design the unfolding transformation.

Therefore, all inputs for the next sequence of cycle need to be changed based on the movement of input which was already applied in parallel form. Each module needed to be modified in terms of inputs to obtain inner pipelining and unfolding with factors two and four. These modifications provided the improvements for the SHA-256 hash function. The number of cycles is reduced based on the number of J factors using an unfolding design technique [3]. Moreover, the throughput of the SHA-256 algorithm is also increased using this technique. The number of cycles was reduced from 66.5 to 35.5 for unfolding factor 2 and for unfolding factor 4 it was reduced from 35.5 to 19.5. These cycle numbers were obtained from waveform simulation results of the design. It reduced due to the structures of SHA-256 design unfolding design change based on different inputs.

By reducing the number of cycles with unfolding transformation techniques, the throughput of the design improved significantly. Furthermore, the performance of the SHA-256 hash function has improved in terms of frequency due to the inner pipelining method. The frequency of SHA-256 unfolding with factor 2 increased significantly compared to conventional design. Even though the latencies were reduced by factor 2, modification for SHA-256 unfolding with factor 4 increase the area implementation compared with two other SHA-256 designs. However, it provided high throughput because of low latencies.

The unfolding factor 2 and 4 architectures were produced by the modified message schedule and compression function of the SHA-256 algorithm. The unfolding technique with factor 2 and 4 had been implemented in this paper. Each block in the message schedule and compression function must be considered for its modifications. The modifications of each of the block in the message schedule and compression function must be considered. Figure 1 and Figure 2 show the block diagrams of Temp\textsubscript{10} and Temp\textsubscript{20}. The following block diagrams of Temp\textsubscript{10} and Temp\textsubscript{20} were the modifications of conventional Temp1 and Temp2. The input sequence of SHA-256 unfolding design remakes the output gave different results. The compression function of the SHA-256 algorithm was added with these equations. Temp\textsubscript{10} consists of $\Sigma 10$, Cho\textsubscript{next e,e,f}, Message, $W_{t,1}$ and Constant $K_{t,1}$; while Temp\textsubscript{20} contains $\Sigma 0$ and Majo\textsubscript{next a,a,b}. These results were obtained using a 32-bit adder. All data inputs were different for each of the blocks in Temp\textsubscript{10} and Temp\textsubscript{20} block diagram architectures.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{Temp10.png}
\caption{Temp\textsubscript{10} Block Diagram Architecture}
\end{figure}

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{Temp20.png}
\caption{Temp\textsubscript{20} Block Diagram Architecture}
\end{figure}
The Figures 3 and Figure 4 show the two architectures for Cho(next_e,e,f) and Majo(next_a,a,b). The AND, NOT and XOR gates are all components of both the architectures, with different structures of implementation. The data inputs for both the architectures were different from conventional equation for function Ch and Maj. The compression function of the SHA-256 algorithm that was used to obtain data input next_e and next_a.

![Figure 3. Cho (next_e,e,f) Function](image1)

![Figure 4. Majo(next_a,a,b) Function](image2)

Figures 5 and Figure 6 illustrate the proposed architectures for \( \Sigma 0 \) o and \( \Sigma 1 \) o, respectively. \( \Sigma 0 \) o was depicted by input next_a while for \( \Sigma 1 \) o the input is next_e. Both architectures had all rotations follow in the right direction with a fixed amount of values. Finally, the final outputs of \( \Sigma 0 \) o and \( \Sigma 1 \) o was obtained by combining all inputs using an XOR gate.

![Figure 5. \( \Sigma 0 \) o(next_a) Architecture](image3)

![Figure 6. \( \Sigma 1 \) o(next_e) Architecture](image4)

Output Temp\(_{2o}\) and Temp\(_{2o}\) was used to calculate new next\(_{eo}\) and next\(_{ao}\). The following equations show the output for next\(_{eo}\) and next\(_{ao}\).

\[
\begin{align*}
\text{next}\_\text{eo} & = c + \text{Temp}_{1o} \\
\text{next}\_\text{ao} & = \text{Temp}_{1o} + \text{Temp}_{2o}
\end{align*}
\]

Figures 7 and 8 show the block diagram architecture for Temp\(_{11}\) and Temp\(_{21}\). For Temp\(_{11}\), the inputs for Ch1 were next\(_{eo}\), next\(_{e}\), and e while for \( \Sigma 11 \), the input was next\(_{eo}\). For Temp\(_{21}\), the input for Maj1 were next\(_{ao}\), next\(_{a}\) and a while for \( \Sigma 01 \), the input was next\(_{ao}\).

![Figure 7. Temp\(_{11}\) Block Diagram](image5)

![Figure 8. Temp\(_{21}\) Block Diagram](image6)

Then, Temp\(_{11}\) and Temp\(_{21}\) were calculated in order to obtain next\(_{e1}\) and next\(_{a1}\). The outputs for both equations are illustrated in the following equation.

\[
\begin{align*}
\text{next}\_\text{e1} & = b + \text{Temp}_{11} \\
\text{next}\_\text{a1} & = \text{Temp}_{11} + \text{Temp}_{21}
\end{align*}
\]
Since unfolding factor 4 needed four parallel executions, it was calculated until Temp\textsubscript{12} and Temp\textsubscript{22}.

![Figure 9. Temp\textsubscript{12} Block Diagram](image)

Figure 9. Temp\textsubscript{12} Block Diagram

![Figure 10. Temp\textsubscript{22} Block Diagram](image)

Figure 10. Temp\textsubscript{22} Block Diagram

Figures 9 and Figure 10 show the output for Temp\textsubscript{12} and Temp\textsubscript{22}. The input for Σ\textsubscript{12} was next_e\textsubscript{1} while for Σ\textsubscript{02} the input was next_a\textsubscript{1}. The sequence of the input was similar to the previous but the data was moved one place. For example, the inputs for Ch2 were next_e\textsubscript{1}, next_e\textsubscript{o} and next_e. For Maj\textsubscript{2}, the inputs are next_a\textsubscript{1}, next_a\textsubscript{o} and next_a. The data for next_e\textsubscript{2} and next_a\textsubscript{2} was calculated based on data from Temp\textsubscript{12} and Temp\textsubscript{22}. The following equation shows the next_e\textsubscript{2} and next_a\textsubscript{2} output.

\[ \text{next}_e\textsubscript{2} = a + \text{Temp}_{12} \] \hspace{1cm} (8)

\[ \text{next}_a\textsubscript{2} = \text{Temp}_{12} + \text{Temp}_{22} \] \hspace{1cm} (9)

Similar to the compression function, the message schedule was modified from previous results. Once the first message schedule was received, the new equation for sigma\textsubscript{00} and sigma\textsubscript{10} and next_w\textsubscript{to} was processed. In order to obtain next_w\textsubscript{t} and next_w\textsubscript{t1}, the input was moved one place. The input message for wt\textsubscript{0} was from message wt\textsubscript{2}. This sequence started from wt\textsubscript{2} until wt\textsubscript{15}. After that, the input followed the sequence of next_w\textsubscript{t} starting from next_w\textsubscript{t} and next_w\textsubscript{t0}. So, we needed to add another W_message as input such as W_message and W_message\textsubscript{1}.

The architectures for both σ\textsubscript{00} and σ\textsubscript{10} functions are shown by Figures 11 and 12. Generating a message schedule for SHA-256 is the main function of these architectures. W\textsubscript{2} was rotated in the right direction with a fixed amount of value for σ\textsubscript{00} while for σ\textsubscript{10}, a different value was used rotate it. The W\textsubscript{2} was right shifted by 3 in σ\textsubscript{00} architecture, and for σ\textsubscript{10} architecture, the W\textsubscript{15} was right shifted by 10.

![Figure 11. σ\textsubscript{00} Architecture](image)

Figure 11. σ\textsubscript{00} Architecture

![Figure 12. σ\textsubscript{10} Architecture](image)

Figure 12. σ\textsubscript{10} Architecture

For unfolding factor 4, σ\textsubscript{01} and σ\textsubscript{11} were calculated with input W3 and next_w\textsubscript{t} respectively. Then, next_w\textsubscript{t1} was obtained by the following Equation (10). Finally, the input for σ\textsubscript{02} and σ\textsubscript{12} were obtained from input W4 and next_w\textsubscript{t0} respectively. The next_w\textsubscript{t0} was derived from Figure 13. The structure of next_w\textsubscript{t2} is shown in Figure 13. This figure shows the final structure of the message schedule for unfolding factor 4. The input data for message W0 started with W4 until W15. The
output sequence of next_wt used the similar method used in unfolding factor 2. The sequence of next wt was next_wt, nextk_wto, next_wt1 and next_wt2.

\[
\text{next_wt1} = W_2 + \sigma_{01} + W_{11} + \sigma_{11}
\]  

\[ (10) \]

Figure 13. Message Schedule of SHA-256 Unfolding factor 4

3. Result and Discussion

The SHA-256 unfolding factor 2 and 4, the proposed SHA-256, were successfully designed and tested. All designs were written in Verilog code and compilation process was done using Altera Quartus II. ModelSim was used to simulate and verify for both functional and timing simulation of the design. Equation (11) was used to calculate the throughput of these designs.

\[
\text{Throughput} = \frac{(512 \times \text{FMax})}{\text{Number of Cycle}}
\]  

\[ (11) \]

The results showed the throughput of SHA-256 with unfolding factor 4 increased significantly compared to other SHA-256 designs. The proposed SHA-256 designs were compared with previous publications as shown in Table 1. Table 1 shows the synthesis and implementation comparison results of other SHA-256 designs in terms of FPGA implementation. By using unfolding transformation, the performance of SHA-256 design in terms of throughput increase significantly. This is because of the inner structure of SHA-256 design was processed in parallel form. This method can improve the performance of SHA-256 design where the number of latencies of the design was reduced. Implementation of SHA-256 design on different devices provided different results. Thus, by choosing appropriate family device for implementation, the better results can be obtained.

<table>
<thead>
<tr>
<th>Design</th>
<th>Device</th>
<th>ALUTs/CLBs</th>
<th>Freq (MHz)</th>
<th>Throughput (Mbps)</th>
</tr>
</thead>
<tbody>
<tr>
<td>SHA-256 Design</td>
<td>Arria II GX</td>
<td>855 ALUTs</td>
<td>228.15</td>
<td>1756.58</td>
</tr>
<tr>
<td>SHA-256 Unfolding Design (factor 2)</td>
<td>Arria II GX</td>
<td>1345 ALUTs</td>
<td>251.07</td>
<td>3621.07</td>
</tr>
<tr>
<td>SHA-256 Unfolding Design (factor 4)</td>
<td>Arria II GX</td>
<td>2064 ALUTs</td>
<td>159.82</td>
<td>4196.30</td>
</tr>
<tr>
<td>SHA-2 [4]</td>
<td>Virtex 5</td>
<td>320 CLBs</td>
<td>218.2</td>
<td>1719</td>
</tr>
<tr>
<td>SHA-2 [4]</td>
<td>Stratix III</td>
<td>795 ALUTs</td>
<td>205.8</td>
<td>1621</td>
</tr>
<tr>
<td>SHA(256,384,512) [5]</td>
<td>Virtex v200pq 240-6</td>
<td>2207 CLBs</td>
<td>74</td>
<td>291</td>
</tr>
<tr>
<td>SHA-256 [6]</td>
<td>Virtex v200pq240</td>
<td>1060 CLBs</td>
<td>83</td>
<td>326</td>
</tr>
<tr>
<td>SHA-256 [7]</td>
<td>Stratix II</td>
<td>2150 ALUTs</td>
<td>143.164</td>
<td>909.816</td>
</tr>
<tr>
<td>SHA-256 [8]</td>
<td>Virtex 5 XC5VFX70T</td>
<td>387 Slices</td>
<td>202.54</td>
<td>1580</td>
</tr>
<tr>
<td>SHA-256[9]</td>
<td>XC2PV-7</td>
<td>755 Slices</td>
<td>174</td>
<td>1370</td>
</tr>
<tr>
<td>SHA-256 [10]</td>
<td>Virtex-II xc2v2000-bf957</td>
<td>1373 Slices</td>
<td>133.06</td>
<td>1009</td>
</tr>
<tr>
<td>SHA-256 [12]</td>
<td>Virtex-4</td>
<td>610 Slices</td>
<td>170.75</td>
<td>1344.98</td>
</tr>
<tr>
<td>SHA-256 [13]</td>
<td>Virtex-5 XC5VFX70T</td>
<td>387 Slices</td>
<td>202.54</td>
<td>1.58</td>
</tr>
<tr>
<td>SHA-256 [13]</td>
<td>Virtex</td>
<td>1534 CLBs</td>
<td>35.1</td>
<td>2077</td>
</tr>
<tr>
<td>SHA-256 [14]</td>
<td>Virtex E</td>
<td>1655 CLBs</td>
<td>36.4</td>
<td>2190</td>
</tr>
<tr>
<td>SHA-256 [15]</td>
<td>Virtex E</td>
<td>-</td>
<td>64.1</td>
<td>2052.1</td>
</tr>
<tr>
<td>SHA-256 [16]</td>
<td>Virtex II</td>
<td>1708 slices</td>
<td>52.1</td>
<td>3100.9</td>
</tr>
<tr>
<td>SHA-256 [17]</td>
<td>Cyclone II</td>
<td>7219 cells</td>
<td>116.24</td>
<td>875.22</td>
</tr>
</tbody>
</table>

Lee Y.K. et al. [17] discovered by using ASIC the implementation of the SHA-256 design provided large area implementation in terms of gates which is 22,025. Even though the throughput of SHA-256 design was high but area implementation is too large. It is better to have balance between area and throughput of SHA-256 design. This paper proposed small area implementation and high throughput of SHA-256 design by using unfolding transformation factor 4. 2064 ALUTs was used by the proposed SHA-256 unfolding factor 4 and 159.82 MHz was the maximum clock frequency of this design. The throughput of this design was improved by increasing the number of unfolding factor. From the table, the result shows that the SHA-256 unfolding design with factor 4 give the highest throughput in terms of FPGA implementation which is 4196.30 Mbps with 159.82 MHz maximum frequency. Consequently, the proposed SHA-256 unfolding factor 2 produced better results in terms of maximum frequency due to the inner pipelining design where 1159 registers were used. The novelty of this paper is the design of SHA-256 using the unfolding transformation with factor 4 can improve the throughput of SHA-256 design. The throughput of the SHA-256 design was improved by using this method due to the small number of latencies compared to the conventional design. The number of clock cycles of SHA-256 unfolding factor 4 design decreased from 66.5 cycles of conventional design to 19.5 cycles of unfolding design. Thus, the high throughput of SHA-256 design was obtained by using the unfolding transformation method.

4. Conclusion

The study successfully completed and tested SHA-256 unfolding factor 2 and 4 designs which were the proposed SHA-256. The area and maximum frequency are comparable to other SHA-256 designs. The proposed SHA-256 unfolding with factor 4 design gave the highest throughput with 4196.30 Mbps based on the comparison with other SHA-256 designs. In conclusion, implementation of unfolding transformation can improve the performance of SHA-256 hash function by reducing the number of cycles where the data generate in parallel transformation. This leads to high throughput of SHA-256 design. The throughput of SHA-256 design also increases significantly by using this methodology. In the future, the Keyed-hash Message Authentication Codes (HMAC) can utilize the proposed SHA-256 design in order to enhance the performance of security design.
Acknowledgements

The authors would like to thank Universiti Malaysia Sarawak (UNIMAS) and Waseda University for providing opportunity and facilities to support this project.

References


IEEE 12th International Multi-Conference on Systems, Signals & Devices (SSD15), (pp. 1-5).


McEvoy, R.P., Crowe, F.M., Murphy, C.C. and Marnane, W.P., 2006, March. Optimisation of the SHA-2 family of hash functions on FPGAs. In IEEE Computer Society Annual Symposium on Emerging VLSI Technologies and Architectures (ISVLSI'06) (pp. 6-pp). IEEE. DOI: 10.1109/ISVLSI.2006.70

binti Suhaili, S. and Watanabe, T., 2017, November. Design of high-throughput SHA-256 hash function based on FPGA. In 2017 6th International Conference on Electrical Engineering and Informatics (ICEEI) (pp. 1-6). IEEE.
Sun, W., Guo, H., He, H. and Dai, Z., 2007, October. Design and optimized implementation of the SHA-2 (256, 384, 512) hash algorithms. In 2007 7th International Conference on ASIC (pp. 858-861). IEEE. DOI: 10.1109/ICASIC.2007.4415766