Iteration Bound Analysis and Throughput Optimum Architecture of SHA-256 (384,512) for Hardware Implementations

Yong Ki Lee[(1)], Herwin Chan[(1)], and Ingrid Verbauwhede[(1),(2)]

[(1)] University of California, Los Angeles, USA
[(2)] Katholieke Universiteit Leuven, Belgium
{jfirst,herwin,ingrid}@ee.ucla.edu

Abstract. The hash algorithm forms the basis of many popular cryptographic protocols and it is therefore important to find throughput optimal implementations. Though there have been numerous published papers proposing high throughput architectures, none of them have claimed to be optimal. In this paper, we perform iteration bound analysis on the SHA2 family of hash algorithms. Using this technique, we are able to both calculate the theoretical maximum throughput and determine the architecture that achieves this throughput. In addition to providing the throughput optimal architecture for SHA2, the techniques presented can also be used to analyze and design optimal architectures for some other iterative hash algorithms.

Key words: SHA-256 (384,512), Iteration Bound Analysis, Throughput Optimum Architecture

1 Introduction

Hash algorithms produce a fixed size code independent of input size of messages. Generated codes from hash algorithms are commonly used for digital signature [1] and message authentication. Since the hash outputs are relatively small compared to the original messages, the hash algorithms take an important roll for computation efficiency. Considering the increasing data amount to store or communicate, the throughput of hash algorithms is an important factor. Common hash algorithms include SHA1, MD5, SHA2 family (SHA256, SHA384 and SHA512) and RMD family (RMD160, RMD256 and RMD320). The SHA2 family of hash algorithms [2] has become of particular interest lately due the official support of the National Institute of Standards and Technology (NIST) in 2002.

Even though many publications were produced to show high throughput architectures of SHA2, there has been no mathematical analysis of the delay bound. In this paper, we analyze the iteration bound analysis of SHA2, which gives us the maximum theoretical throughput achievable by the algorithm. Knowing the iteration bound not only allows the designer a goal to design towards, but also signals the designer when an optimal architecture has been achieved.

The remainder of this paper is organized as follows. We start with reviewing related work in Section 2 and introduce the iteration bound analysis and transformations in Section 3. In Section 4, we analyze the iteration bound of the SHA2 family of hash algorithms and show the procedure to design throughput optimum architectures. Some comments for implementation and synthesis results are given in Section 5 followed by the conclusion in Section 6.

2 Related Works

The most common techniques for high throughput hardware implementations of hash algorithms are pipelining, unrolling and using Carry Save Adder (CSA). Pipelining techniques are used in [4–7], unrolling techniques are used in [7–10], and CSA techniques are used in [4–7, 10, 11]. Some other interesting implementations can be found in [12, 13].

Even though there are many published papers, the mathematical analysis of the iteration bound has rarely been performed. For example, even though the iteration bound for SHA2 was achieved in [4], no analysis or proof of optimality was given. Since there is no proof of theoretical optimality and systematic approach to achieve the optimality, many architectures seem to count on the designers' intuition. Actually the work of [4] achieved the theoretical optimum after another work [5]. Therefore, this work will not only prevent a futile attempt to design architecture achieving better throughput than the theoretical optimum but also will guide designers to achieve the theoretical optimum throughput in MD4-based hash algorithms.

3 The Iteration Bound Analysis and Transformations

The SHA2 family of hash algorithms are iterative algorithms, which means the output of one iteration is the input of the next. We use a Data Flow Graph (DFG) to represent dependencies. We continuously apply the techniques of retiming and unfolding to achieve the iteration bound. Even though the optimized SHA2 family of hash algorithms requires only the retiming transformations, both transformations are briefly discussed for a better understanding of the analysis technique. Some of the MD4-based hash algorithms may require the unfolding transformation. A more detailed discussion of the iteration bound and the transformations can be found in [3].

3.1 DFG Representation

The mathematical expression of our example is given in Eq. 1. A and B are variables which are stored in registers and the indices of the variables represent the number of iterations of the algorithm. C_1 and C_2 are some constants. According to the equation, the next values of variables are updated using the current values

of variables and constants. This type of equations is very common in MD4-based hash algorithms.

$$A(n+1) = A(n) + B(n) * C_1 * C_2 \qquad (1)$$
$$B(n+1) = A(n)$$

The DFG of Eq. 1 is shown in Fig. 1. Box A and B represent registers which give the output at cycle n, and circles represent some functional nodes which perform the given functional operations. A D on edges represents an *algorithmic* delay, i.e. a delay that cannot be removed from the system. Next to algorithmic delays, nodes also have functional delays. We express the functional delays, i.e. the delays to perform the given operations, of $+$ and $*$ as $Prop(+)$ and $Prop(*)$ respectively. The binary operators, $+$ and $*$, can be arbitrary operators but we assume $Prop(+) < Prop(*)$ in this example. The iteration bound analysis starts with an assumption that any functional operation is atomic. This means that a functional operation can not be split or merged into some other functional operations.

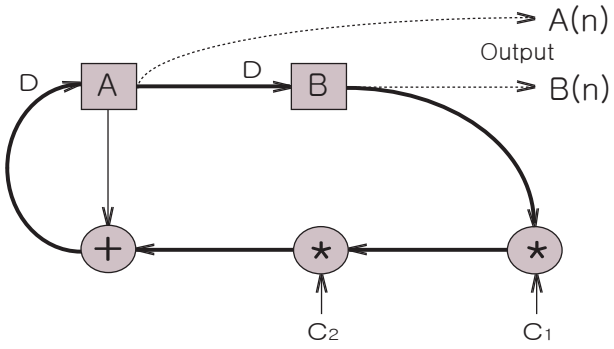


Fig. 1. An example of DFG

3.2 The Iteration Bound Analysis

A loop is defined as a path that begins and ends at the same node. In the DFG in Fig. 1, $A \longrightarrow + \xrightarrow[D]{} A$ and $A \xrightarrow[D]{} B \longrightarrow * \longrightarrow * \longrightarrow + \xrightarrow[D]{} A$ are the loops. The loop calculation time is defined as the sum of the functional delays in a loop. If t_l is the loop calculation time and w_l is the number of algorithmic delays in the l-th loop, the l-th loop bound is defined as t_l/w_l. The iteration bound is defined as follows:

$$T_\infty = \max_{l \in L} \left\{ \frac{t_l}{w_l} \right\} \qquad (2)$$

where L is the set of all possible loops. The iteration bound creates a link between the arithmetic delay and the functional delay. It is the theoretical limit of a DFG's delay bound. Therefore, it defines the maximally attainable throughput. Please note that every loop needs to have at least one algorithmic delay in the loop otherwise the system is not causal and cannot be executed.

Since the loop marked with bold line has the maximum loop delay assuming that $Prop(+) < Prop(*)$, the iteration bound is as follows:

$$\begin{aligned} T_\infty &= \max\left\{Prop(+), \frac{Prop(+) + 2 \times Prop(*)}{2}\right\} \\ &= \frac{Prop(+) + 2 \times Prop(*)}{2} \end{aligned} \tag{3}$$

This means that a critical path delay in this DFG can not be less than this iteration bound. The critical path delay is defined as the maximum calculation delay between any two consecutive algorithmic delays, i.e. D's. In our example (Fig. 1), the critical path delay is $Prop(+) + 2 \times Prop(*)$ which is larger than the iteration bound. The maximum clock frequency (and thus throughput) is determined by the critical path (the slowest path). The iteration bound is a theoretical lower bound on the critical path delay of an algorithm. We use the retiming and unfolding transformations to reach this lower bound.

3.3 The Retiming Transformation

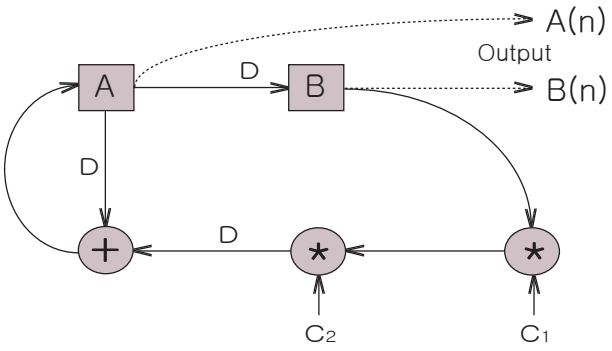
The minimum critical path delay that can be possibly achieved using the retiming transformation is shown in Eq. 4.

$$\lceil T_\infty \rceil = \left\lceil \frac{Prop(+) + 2 \times Prop(*)}{2} \right\rceil = Prop(+) + Prop(*) \tag{4}$$

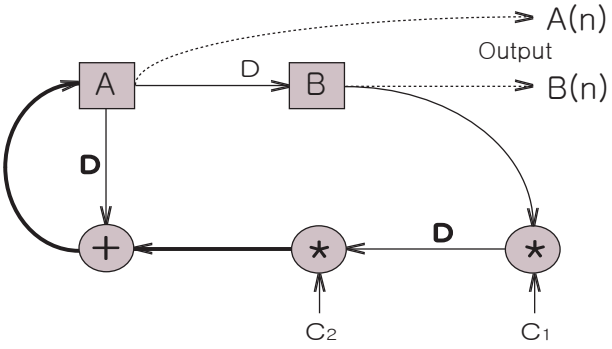
Assuming that a functional node can not be split into multiple parts, $\lceil \cdot \rceil$ is the maximum part when $Prop(+) + 2 \times Prop(*)$ is evenly distributed into N parts, where N is the number of algorithmic delays in a loop. This is denoted by the 2 in our example and sits in the denominator. Since the total delay $Prop(+) + 2 \times Prop(*)$ can be partitioned into one delay $Prop(+) + Prop(*)$ and the other delay $Prop(*)$, the attainable delay bound by the retiming transformation is $Prop(+) + Prop(*)$.

The retiming transformation modifies a DFG by moving algorithmic delays, i.e. D's, through the functional nodes in the graph. Delays of out-going edges can be replaced with delays from in-coming edges and vice versa. Fig. 2 shows the retiming transformation steps performed on Fig. 1.

Based on the + node in Fig. 1, the delay of the out-going edge is replaced with delays of the in-coming edges resulting in Fig. 2(a). Note that the out-going edges and the in-coming edges must be dealt as a set. By performing one more retiming transformation based on the left $*$ node in Fig. 2(a), we obtain the DFG of Fig. 2(b). Therefore, the critical path becomes the path in bold between the two bolded D's in Fig. 2(b) and its delay is reduced to $Prop(+) + Prop(*)$ which is the same as Eq. 4. However, the iteration bound still has not been met.



(a) Retimed based on +



(b) Retimed based on ∗

Fig. 2. Retiming Transformation

3.4 The Unfolding Transformation

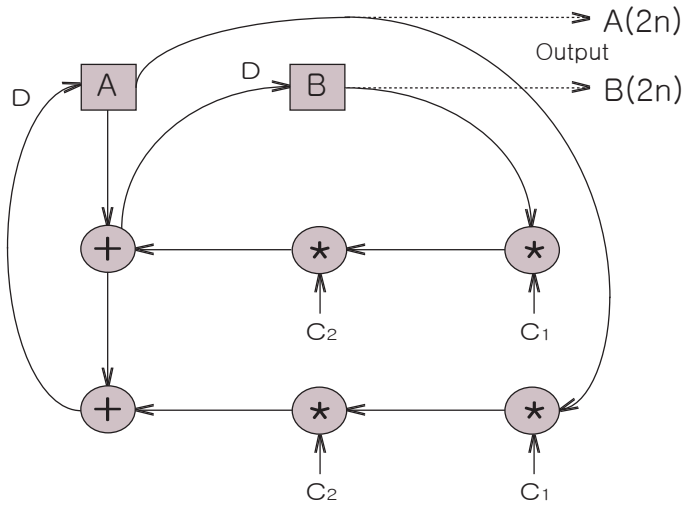
The unfolding transformation improves performance by calculating several iterations in a single cycle. For the unfolding transformation we expand the Eq. 1 by representing two iterations at a time, which results in Eq. 5.

$$\begin{aligned} A(n+2) &= A(n+1) + B(n+1) * C_1 * C_2 \\ &= A(n) + B(n) * C_1 * C_2 + A(n) * C_1 * C_2 \\ B(n+2) &= A(n+1) = A(n) + B(n) * C_1 * C_2 \end{aligned} \tag{5}$$

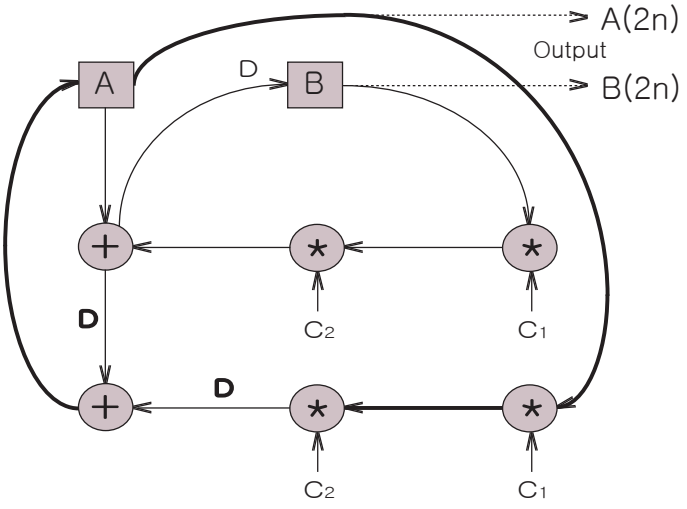
Note that now $A(n+2)$ and $B(n+2)$ are expressed as a function of $A(n)$ and $B(n)$. By introducing a temporary variable Tmp, Eq. 5 can be simplified into Eq. 6.

$$\begin{aligned} Tmp(n) &= A(n) + B(n) * C_1 * C_2 \\ A(n+2) &= Tmp(n) + A(n) * C_1 * C_2 \\ B(n+2) &= Tmp(n) \end{aligned} \tag{6}$$

By doubling the number of functional nodes, we are able to unfold the DFG by a factor of two (Fig. 3(a)). Box A and B now give the outputs of every



(a) Unfolding Transformation



(b) Retiming Transformation

Fig. 3. Unfolding and Retiming Transformation

second iteration. By applying the retiming transformation to the unfolded DFG, the resulting critical path becomes the path in bold between the two bolded D's which is $D \rightarrow + \rightarrow A \rightarrow * \rightarrow * \rightarrow D$ (Fig. 3(b)). Therefore, the critical path delay is $Prop(+) + 2 \times Prop(*)$. Due to the unfolding factor of two, the normalized critical path delay, $\hat{T}$, can be calculated by dividing the critical path delay by two as shown in Eq. 7.

$$\hat{T} = \frac{Prop(+) + 2 \times Prop(*)}{2} = T_{\infty} \tag{7}$$

This final transformation results in an architecture that achieves the iteration bound of the example DFG (Fig. 1).

Now the only remaining step is the implementation of the resulting DFG. Note that some of the square nodes are not any more paired with an algorithmic delay, which can be seen in Fig. 3(b). The explanation about how this issue is dealt with during implementation will be given in Section 5 where we synthesize the SHA2 family hash algorithms.

4 Iteration Bound Analysis and Throughput Optimum Architecture of SHA2

$$
\begin{aligned}
&Ch(x,y,z) = (x \wedge y) \oplus (\neg x \wedge z)\\
&Maj(x,y,z) = (x \wedge y) \oplus (x \wedge z) \oplus (y \wedge z)\\
&\Sigma_0^{\{256\}}(x) = ROTR^2(x) \oplus ROTR^{13}(x) \oplus ROTR^{22}(x)\\
&\Sigma_1^{\{256\}}(x) = ROTR^6(x) \oplus ROTR^{11}(x) \oplus ROTR^{25}(x)\\
&\sigma_0^{\{256\}}(x) = ROTR^7(x) \oplus ROTR^{18}(x) \oplus SHR^3(x)\\
&\sigma_1^{\{256\}}(x) = ROTR^{17}(x) \oplus ROTR^{19}(x) \oplus SHR^{10}(x)
\end{aligned}
$$

(a) SHA-256 Functions

$$
W_t = \begin{cases} M_t^{(i)} & 0 \le t \le 15 \\ \sigma_1^{\{256\}}(W_{t-2}) + W_{t-7} + \sigma_0^{\{256\}}(W_{t-15}) + W_{t-16} & 16 \le t \le 63 \end{cases}
$$

(b) SHA-256 Expend Computation

$$
\begin{aligned}
&T_1 = h + \Sigma_1^{\{256\}}(e) + Ch(e,f,g) + K_t^{\{256\}} + W_t\\
&T_2 = \Sigma_0^{\{256\}}(a) + Maj(a,b,c)\\
&h = g\\
&g = f\\
&f = e\\
&e = d + T_1\\
&d = c\\
&c = b\\
&b = a\\
&a = T_1 + T_2
\end{aligned}
$$

(c) SHA-256 Compress Computation

Fig. 4. SHA-256 Hash Computation

The SHA2 family of hash algorithms is composed of three parts: the padding, expander and compressor [2]. The padding extends an input message to be a whole number of 512-bit (for SHA-256) or 1024-bit (for SHA-384 and SHA-512) message blocks. The expander enlarges input messages of 16 words into 64 (for

SHA-256) or 80 (for SHA-384 or SHA-512) words. The compressor encodes the expanded message into 256, 384 or 512 bits depending on the algorithm. For one message block, the required iterations are 64 (for SHA-256) or 80 (for SHA-384 or SHA-512).

In this paper, we consider only the implementation of the expander and compressor. Though the expander can be performed before the compressor, we chose to implement it to perform dynamically during compression in order to increase the overall throughput and minimize the gate area.

Fig. 4 describes the SHA-256 algorithm on which a DFG will be derived based. Since all the SHA2 family hash algorithms have the same architecture except for input, output and word sizes, constants, non-linear scrambling functions, i.e. Σ_0, Σ_1, Maj, Ch, σ_0 and σ_1, and the number of the iterations, they can be expressed in the same DFG.

4.1 DFG of SHA2 Compressor

Since within one iteration the order of additions in SHA2 does not affect the results, there are several possible DFG's. For example, $(a+b)+c$ and $(b+c)+a$ are equivalent in mathematics but will have different DFG's. As a starting point, the DFG having the minimum iteration bound must be chosen, transformations are then performed to find the architecture that achieves this bound. In SHA2 compressor, since there are only 7 adders, finding a DFG having the minimum iteration bound is not difficult as long as we understand how to calculate the iteration bound.

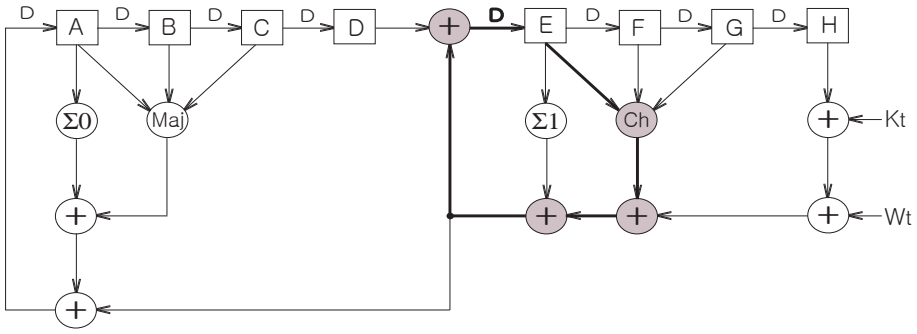


Fig. 5. Basic SHA2 Compressor DFG

The DFG in Fig. 5 is a straightforward DFG. The shaded loop indicates the loop having the largest loop bound and gives the following iteration bound.

$$T_{\infty}^{(5)} = \max_{l \in L} \left\{ \frac{t_l}{w_l} \right\} = 3 \times Prop(+) + Prop(Ch) \tag{8}$$

However, by reordering the sequence of additions, the DFG of Fig. 6 can be obtained which has the smallest iteration bound. As we assume that $Prop(\Sigma 0) \approx Prop(Maj) \approx Prop(\Sigma 1) \approx Prop(Ch)$, the two bolded loops have the same maximum loop bound. Since the loop bound of the left hand side loop cannot be reduced further, no further reduction in the iteration bound is possible. Therefore, the iteration bound of Fig. 6 is as follows.

$$T_{\infty}^{(6)} = \max_{l \in L} \left\{ \frac{t_l}{w_l} \right\} = 2 \times Prop(+) + Prop(Ch) \tag{9}$$

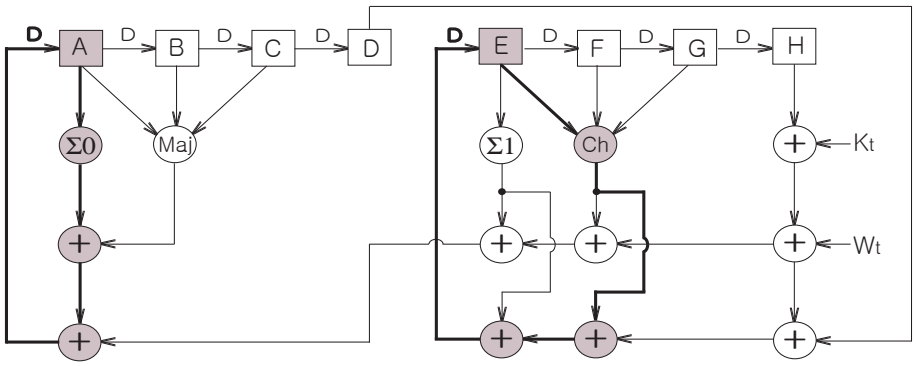


Fig. 6. Optimized SHA2 Compressor DFG

If we assume that any operation in the DFG cannot be merged or split into other operations, the iteration bound of SHA2 is Eq. 9. However, if we are allowed to use a Carrier Save Adder (CSA), we can substitute two consecutive adders with one CSA and one adder. Since CSA requires less propagation delay than an adder, we replace adders with CSA's if it is possible. The resulting DFG is shown in Fig. 7. Note that some of the adders are not replaced with CSA since doing so would increase the iteration bound. Therefore, the final iteration bound is achieved as Eq. 10.

$$T_{\infty}^{(7)} = \max_{l \in L} \left\{ \frac{t_l}{w_l} \right\} = Prop(+) + Prop(CSA) + Prop(Ch) \tag{10}$$

In the next step, we perform transformations. Since there is no fraction in the iteration bound, we do not need the unfolding transformation. Only the retiming transformation is necessary to achieve the iteration bound. The retimed DFG achieving the iteration bound is depicted in Fig. 8. Note that the indexes of K_{t+2} and W_{t+3} are changed due to the retiming transformation. In order to remove the ROM access time for K_{t+2}, which is a constant value from ROM, we place an algorithmic delay, i.e. D, in front of K_{t+2}. This does not change the function.

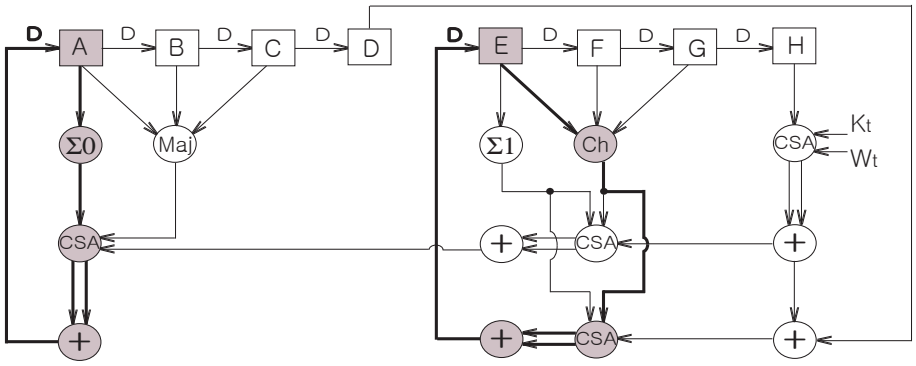


Fig. 7. Optimized SHA2 Compressor DFG with CSA

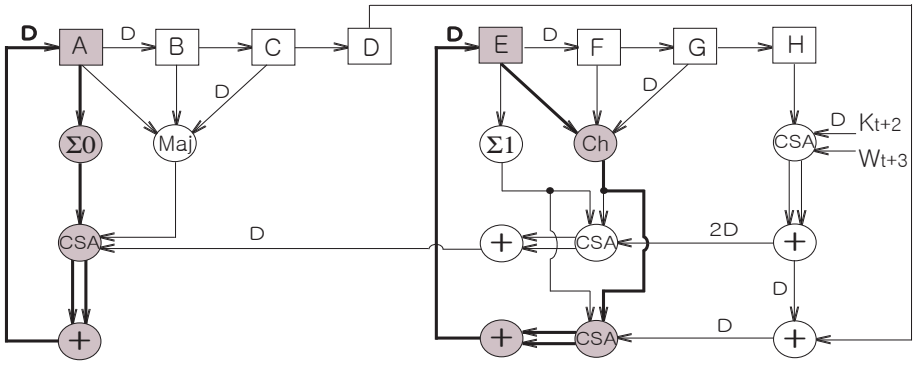


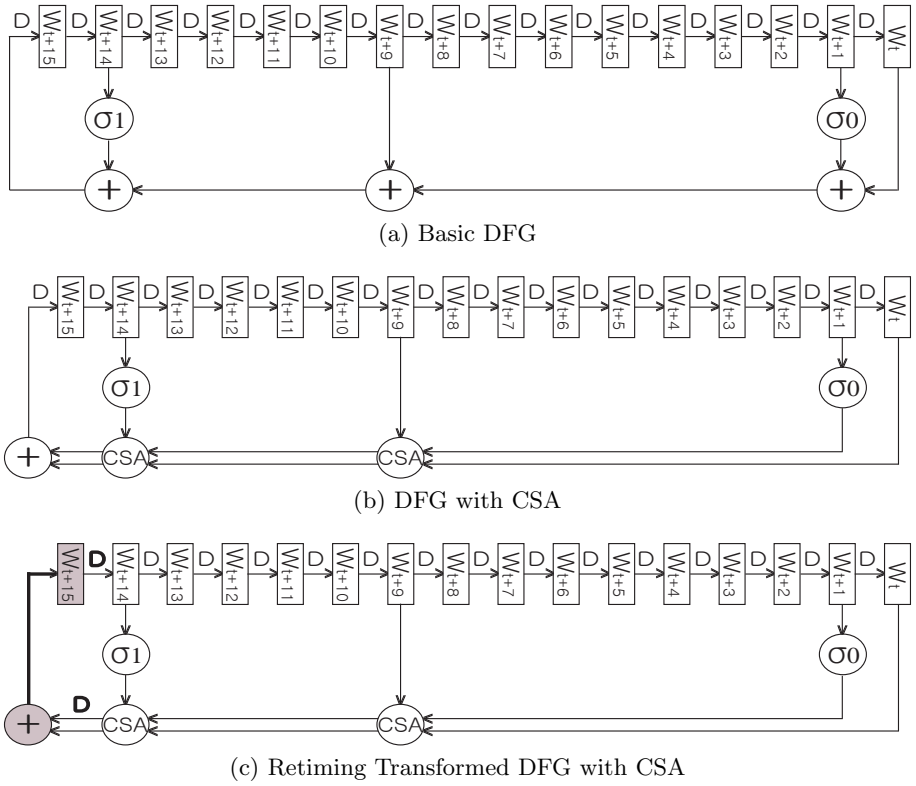
Fig. 8. Final SHA2 Compressor DFG with Retiming Transformation

4.2 DFG of SHA2 Expander

A straightforward DFG of the SHA2 expander is given in Fig. 9(a). Even though the iteration bound of the expander is much less than the compressor, we do not need to minimize the expander's critical path delay less than the compressor's iteration bound (the throughput is bounded by the compressor's iteration bound). Fig. 9(b) shows a DFG with CSA, and Fig. 9(c) shows a DFG with the retiming transformation where the critical path delay is $Prop(+)$.

5 Implementation and Synthesis Results

In the DFG of Fig. 8, some of the register values, i.e. A, B, ..., H, are no longer paired with an algorithmic delay D. For example, there is no algorithmic delay between registers F and H. Therefore, the values of H will be the same as F except for the first two cycles: in the first cycle, the value of H should be the initialized value of H according to the SHA2 algorithm; in the second cycle the



(a) Basic DFG

(b) DFG with CSA

(c) Retiming Transformed DFG with CSA

Fig. 9. SHA2 Expander DFG

value of H should be the initialized value of G. Therefore, the value of F will be directly used as an input of the following CSA.

Another register management problem is found in the path from the register H to the register A which includes four algorithmic delays. Therefore, register A has to hold its initial value until an effective value of H reaches to the register A, which means the register A must hold the first three cycles and then it can update with a new value. This causes overhead of three extra cycles. Therefore the total number of cycles required for one message block is the number of iterations plus one cycle for initialization and finalization plus three overhead cycles due to the retiming transformation, which results in 68 cycles for SHA256 and 84 cycles for SHA384 and SHA512.

We synthesized SHA2 for an ASIC by Synopsys Design Vision using a $0.13\mu m$ standard cell library whose results and a comparison with other works are shown in Table 1. The throughputs are calculated using the following equation.

$$Throughput^{256} = \frac{Frequency}{\#\ of\ Cycles} \times (512\ bits) \qquad (11)$$
$$Throughput^{384,512} = \frac{Frequency}{\#\ of\ Cycles} \times (1024\ bits)$$

Table 1. Synthesis Results and Comparison of SHA2 Family Hashes

	Algorithm	Technology (ASIC)	Area (Gates)	Frequency (MHz)	Cycles	Throughput (Mbps)
[14]	SHA256	$0.18\mu m$	22,000	200	65	1,575
[13]	SHA256	$0.13\mu m$	15,329	333.3	72	2,370
	SHA384/512		27,297	250.0	88	2,909
[4]	SHA256	$0.13\mu m$	N/A	>1,000	69	>7,420
Our Results	SHA256	$0.13\mu m$	22,025	793.6	68	5,975
	SHA384/512		43,330	746.2	84	9,096

*All our proposals include all the registers and ROM.

Note that our purpose of the synthesis is not to beat the throughput record but to verify our architecture by checking the correct hash outputs and the actual critical path. Since our HDL programming is done at register transfer level and we have mostly concentrated on optimizing micro-architecture rather than focusing lower-level optimization, some other reported results, e.g. [4], achieve better performance with the same iteration bound delay. However the iteration bound analysis still determines the optimum high level architecture of an algorithm.

6 Conclusion

We analyzed the iteration bound of the SHA-256 (384,512) hash algorithms and showed architectures achieving the iteration bound. Since the iteration bound is a theoretical limit, there will be no further throughput optimization in micro-architecture level. We also synthesized our design to verify the correctness of our architecture design. Moreover, we illustrated detailed steps from a straightforward DFG to a throughput optimized architecture. This approach will guide how to optimize some other iterative hash algorithms in throughput.

Acknowledgments. This work is supported by NSF CCF-0541472, FWO and funds from Katholieke Universiteit Leuven.

References

1. Digital Signature Standard. National Institute of Standards and Technology. Federal Information Processing Standards Publication 186-2. http://csrc.nist.gov/publications/fips/fips186-2/fips186-2-change1.pdf

2. Secure Hash Standard. National Institute of Standards and Technology. Federal Information Processing Standards Publication 180-2, http://csrc.nist.gov/publications/fips/fips180-2/fips180-2.pdf
3. K.K. Parhi: VLSI Digital Signal Processing Systems: Design and Implementation. Weley (1999) 43–61 and 119–140
4. L. Dadda, M. Macchetti and J. Owen: An ASIC design for a high speed implementation of the hash function SHA-256 (384, 512). ACM Great Lakes Symposium on VLSI (2004) 421–425
5. L. Dadda, M. Macchetti and J. Owen: The design of a high speed ASIC unit for the hash function SHA-256 (384, 512). Proceedings of the conference on Design, Automation and Test in Europe (DATE'04). IEEE Computer Society (2004) 70-75
6. M. Macchetti and L. Dadda: Quasi-pipelined hash circuits. Proceedings of the 17th IEEE Symposium on Computer Arithmetic (ARITH'05) (2005) 222–229
7. R. P. McEvoy, F. M. Crowe, C. C. Murphy and W. P. Marnane: Optimisation of the SHA-2 Family of Hah Functions on FPGAs. Proceedings of the 2006 Emerging VLAI Technologies and Architectures (ISVLSI'06) (2006) 317–322
8. H. Michail, A.P. Kakarountas, O. Koufopavlou and C.E. Goutis: A Low-Power and High-Throughput Implementation of the SHA-1 Hash Function. IEEE International Symposium on Circuits and Systems (ISCAS'05) (2005) 4086–4089
9. F. Crowe, A. Daly and W. Marnane: Single-chip FPGA implementation of a cryptographic co-processor. Proceedings of the International Conference on Field Programmable Technology (FPT'04) (2004) 279–285
10. R. Lien, T. Grembowski and K. Gaj: A 1 Gbit/s partially unrolled architecture of hash functions SHA-1 and SHA-512. CT-RSA 2004, Lecture Notes in Computer Science, Vol. 2964. Springer-Verlag (2004) 324–338
11. Y. Ming-yan, Z. Tong, W. Jin-xiang and Y. Yi-zheng: An Efficient ASIC Implementation of SHA-1 Engine for TPM. The 2004 IEEE Asia-Pacific Conference on Circuits and Systems (2004) 873–876
12. T. S. Ganesh and T. S. B. Sudarshan: ASIC Implementation of a Unified Hardware Architecture for Non-Key Based Cryptographic Hash Primitives. Proceedings of the International Conference on Information Technology: Coding and Computing (ITCC'05) (2005) 580–585
13. A. Satoh and T. Inoue: ASIC-Hardware-Focused Comparison for Hash Functions MD5, RIPEMD-160, and SHS. Proceedings of the International Conference on Information Technology: Coding and Computing (ITCC'05) (2005) 532–537
14. Helion IP Core Products. Helion Technology. http://heliontech.com/core.htm