

The Design of a High Speed ASIC Unit for the Hash Function SHA-256 (384, 512)

Luigi Dadda
ALaRI-USI, Lugano, Switzerland
Politecnico di Milano, Milan, Italy
dadda@alari.ch

Marco Macchetti
Politecnico di Milano, Milan, Italy
macchett@elet.polimi.it

Jeff Owen
ST Microelectronics NV, Manno, Switzerland
jefferson.owen@st.com

Abstract

After recalling the basic algorithms published by NIST for implementing the hash functions SHA-256 (384, 512), a basic circuit characterized by a cascade of full adder arrays is given. Implementation options are discussed and two methods for improving speed are exposed: the delay balancing and the pipelining. An application of the former is first given, obtaining a circuit that reduces the length of the critical path by a full adder array. A pipelined version is then given, obtaining a reduction of two full adder arrays in the critical path. The two methods are afterwards combined and the results obtained through hardware synthesis are exposed, where a comparison between the new circuits is also given.

1. Introduction

A Hash function compresses a string of bits of arbitrary length to a string of fixed length. It produces therefore a *fingerprint* of a message, or a picture, or other block of data, that can be used in assuring message content integrity or data origin authentication. A hash function must satisfy some basic requirements to be considered secure: first, it must be hard to invert, e.g. given a hash value it must be computationally infeasible to find a message that produces exactly this hash value; second, given a message, it must be computationally infeasible to find another message which produces the same hash value. Sometimes it is also required that it must be computationally infeasible to find two random messages which produce the same hash value. Given these properties, the hash calculation becomes an essential service to be provided in electronic mail, e-commerce, financial transactions, and software distribution.

A number of algorithms have been proposed and used so far. We refer here to the SHA (Secure Hash Algorithm) developed by the National Institute of Standards and Technology (NIST) who published in 2002 new versions of the algorithm called SHA-256, SHA-384, SHA-512 [6]. These algorithms differ in the structure from the previous standard SHA-1 and the dimension of the message digest is equal respectively to 256, 384 and 512 bits.

The purpose of this paper is to present some hardware implementations of the above algorithms, with the scope of obtaining a high speed of computation. We have devised means to reach the scope that, to our knowledge, are new and efficient. We do not yet reach the level of a physical layout of the circuit, having proved the functionality and the efficiency at the level of simulation and hardware synthesis, using modern design tools. The motivation of this work is the need to match the speed of computation (e.g. within servers) to the transmission speed achieved in fiber optic links already installed today (40 Gbits/sec).

2. The basic algorithm and the corresponding schemes

The description of the SHA-256, SHA-384 and SHA-512 algorithms can be found in [6]. The corresponding scheme can easily be drawn (see Fig. 1 and Appendix). We will concentrate here on the algorithm's inner part; we suppose that message partitioning and padding is done in software at the system level, and we suppose that the intermediate hashes are accumulated in a bank of registers that is not shown in the Figures for clarity. Moreover, all the optimizations that we apply to the SHA-256 algorithm are easily extendable to SHA-384 and SHA-512, since it is sufficient to simply extend the dimension of the word from 32 to 64 bits.

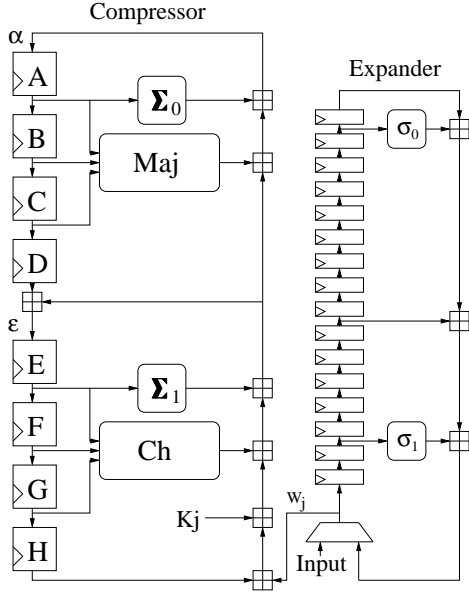


Figure 1. Block diagram of the SHA256 algorithm

The most problematic characteristic of the algorithm's core is the need to add (modulo 2^{32}) several numbers to obtain two main functions which we call α and ϵ . In order to reduce the complexity and to reach a higher operation speed, it is possible to replace a number of adders modulo 2^{32} with a sequence of carry-free adders, composed with full adder arrays, *FAA*'s; this solution has been applied to the SHA-1 algorithm in [5], and to the SHA-512 algorithm in [4]. However, these two solutions, together with [7] and [8], are targeted to reconfigurable devices and consequently optimize the implementation according to the peculiarities of the target FPGA platform. Here we want to give guidelines to build a fast hardware implementation of the SHA-2 family in custom silicon.

We assume the scheme of Fig. 2 as a starting point for our discussion. As we will see, all the proposed optimizations will be applied to this *canonical form* of the circuit; this provides a good common ground to compare implementation techniques and refinements. The scheme is equivalent to the preceding one, using only one carry-propagating-adder per function (actually a carry-look-ahead adder, or *CLA*). A third *CLA* must be used in the connected Expander, whose task is to provide an input to the Compressor. As we can see the Expander has been connected to the Compressor by taking the input W_{ej} directly from the shift register. This solution exploits the fact that the calculation in the Compressor can be delayed by one clock cycle, decreasing the length of the critical path.

We will refer to Fig. 2 as the basic scheme.

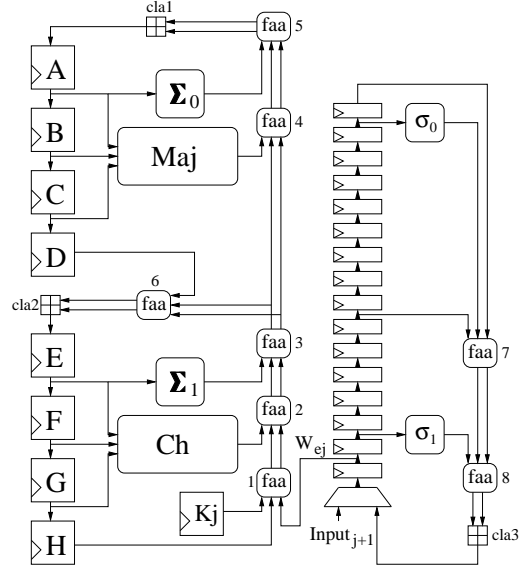


Figure 2. The proposed basic scheme, using Carry-Save-Adders

2.1. The basic scheme's critical path

It can be easily found by inspection that the critical path of the Compressor in the basic scheme is given by:

$$faa1 \Rightarrow faa2 \Rightarrow faa3 \Rightarrow faa4 \Rightarrow faa5 \Rightarrow cla1$$

Its delay is therefore equal to: $\text{delay}(\text{CLA}) + 5 \cdot \text{delay}(\text{FAA})$. The critical path of the Expander is:

$$\sigma_0 \Rightarrow faa7 \Rightarrow faa8 \Rightarrow cla3 \Rightarrow \text{mux}$$

which is considerably shorter. The σ_0 delay is comparable to the delay of a *FAA*. The main purpose of this paper is to present methods that could be used to implement a Compressor with a comparable delay.

2.2. Other basic schemes

The basic scheme defined in Fig. 2 has a simple and regular structure. Other schemes for the computation of α and ϵ can be defined, using various forms of *parallel counters* [2] [3]. It is worth noting, however, that the use of full adder arrays profits from the fact that most of the commonly used custom silicon libraries offer well-optimized full adder designs; moreover, the techniques illustrated in Section 3 cannot be applied if we use these complex components.

The example in Fig. 3 is a tree-like network of *FAA*'s characterized by a delay equal to: $\text{delay}(\text{CLA}) + 4 \cdot \text{delay}(\text{FAA})$ for both functions. The number of *FAA*'s is eight, while they are six in Fig. 2, with a larger delay due to one more *FAA* in the critical path.

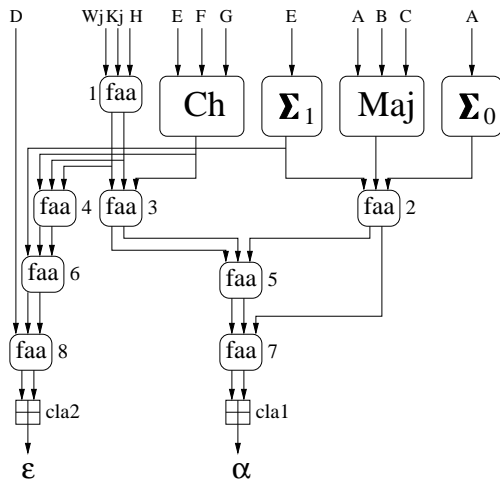


Figure 3. A network of faa's for computing functions α and ϵ

Other circuits can be implemented that make use of: (1) trees of CLA adders; (2) trees of FAA's and two CLA adders; (3) simple parallel counters with 7, 6, and 5 inputs; (4) complex (2-columns) parallel counters. For lack of space, those circuits will not be shown here: we give some results in Table 1 derived from the simulation and hardware synthesis on the STMicroelectronics HCMOS8 technology library, featuring $0.18\mu\text{m}$ process and 1.8V core voltage. Note that the area and timing values refer only to the chain of adders (excluding Ch, Maj, Σ_0 and Σ_1), and that the registers setup and hold timings are not included. We can see that the best area \times delay product is reached when only FAA's are used together with CLA's. For more details, see [1].

3. Further improvements of the critical path

Two methods will be applied in the following:

1. Pipelining: a well known and used method. It offers, nevertheless, some difficulties when applied to our case: it will be shown how to overcome them with a *quasi-pipelining*.
2. Delay balancing: consists in displacing some functional units from the critical path to a shorter path, so that the delay of the critical path is decreased and the delay of the chosen secondary path is correspondingly increased. The ideal case would be to make the two delays identical: this, in practice is not possible, but in any case the two delays will be made more equal. The functionality of the whole system should obviously not be affected.

Components used	Delay	Area	Area \times delay
(7,3)-(3,2)-cla	0.99 ns	146870	145401
(5,3)-(3,2)-cla	0.95 ns	92770	88131
(3,2)-cla #1	1.04 ns	64646	67232
(3,2)-cla #2	1.18 ns	51748	61063
(4,4; 4)-(3,2)-cla	1.41 ns	69512	98012
cla #1	1.39 ns	71323	99139
cla #2	1.71 ns	63454	69799

Table 1. Other implementations of the adder chain

3.1. Delay balancing: first application

Let us consider the circuit in Fig. 4. It is derived from Fig. 2 circuit by applying delay balancing method to register A. The adder *cla1* is moved and A is replaced with A', A'', whose inputs are fed with the two outputs of *faa5*, previously feeding *cla1*. The latter is connected to the A', A'' outputs: its output is identical to the output of A in the original circuit.

The previous critical path starting with *faa1* includes now *faa2*, *faa3*, *faa4* and *faa5*, but does not contain *cla1*. Its delay is therefore $5 \times \text{delay}(\text{FAA})$. Note that the delay in a CLA is considerably greater (about 3 times) than in a FAA. The paths starting with *cla1* continue in

$$\text{Maj} \Rightarrow \text{faa4} \Rightarrow \text{faa5}$$

or in

$$\Sigma_0 \Rightarrow \text{faa5}$$

The longest critical path ends with *cla2*, preceded by *faa6*, *faa3*, *faa2* and *faa1* or Ch (comparable in delay with *faa1*), with total delay approximately given by $\text{delay}(\text{CLA}) + 4 \times \text{delay}(\text{FAA})$. This means a critical path shorter (one less full adder array) than the one of the standard circuit of Fig. 2. In the Figure, the Expander has been removed for clarity.

The functionality is the same as the original scheme. The initial hash value for A is loaded in A' (A'' is cleared). The final hash value is taken from the output of *cla1*. It can be easily verified that all the other paths have a shorter length. It can also be shown that the application of the balancing method to the register E would not improve the delays of the paths involved.

The balancing method could also be applied to tree-like schemes, e.g. the Fig. 3 scheme. However, in this specific case there would be no benefit; in fact, *cla1* would be relocated before Maj and Σ_0 : the paths starting with *faa1*, Ch and Σ_1 would diminish their delay, but the paths starting

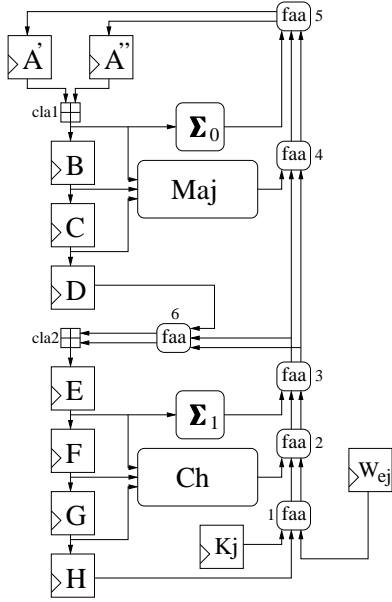


Figure 4. Scheme using the delay balancing principle in the paths to A

with *cla1* (now on top of *Maj* and Σ_0) would keep their original value. Similar result would be obtained by moving *cla2* before *Ch* and Σ_1 .

3.2. Pipelining: first application

A pipelined version of the circuit is shown in Fig. 5. It differs from the standard circuit of Fig. 2 as shown in the following. Two multiplexers have been introduced: *mux1* (*mux1*) selects H or G (D or C) depending on the value of the selector, where the initial value gives H (D) as output. In the bottom section *j* the variables W_{ej} , K_j and the output of *mux2* are first reduced to two in *faa1* and then added with *cla1*, the result feeding register M. Another (double) register L is inserted between *faa2* and *faa4*.

The role of the registers M and L is to define three pipe-sections: *j*, *j-1*, *j-2*. These sections are separated by one clock cycle and one additional clock cycle separates them from the Expander; this of course implies the existence of circuits to control latching, whose structure will determine the operations performed during the first three clock cycles, as described in the following.

- *j* = 0. A–H are loaded with the initial Hash values: A(0), B(0), ..., H(0); *mux1* outputs D(0) which is not yet used; *mux2* outputs H(0) which is being used. The registers M and L are cleared. W_{e0} , K_0 are the first input values.

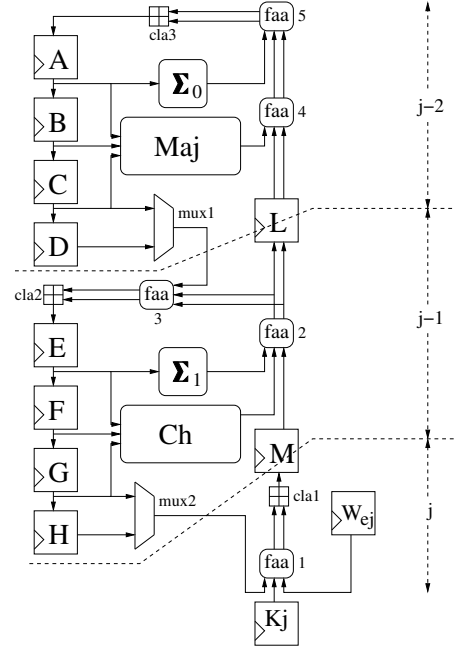


Figure 5. Scheme using pipelining

- *j* = 1. A–H remain unchanged; *mux1* outputs D(0) which is being used and *mux2* outputs G(0) which is being used. M gets its first value, while L is still cleared. W_{e1} , K_1 are the second input values.
- *j* = 2. A–D remain unchanged while E–H get their first new values; *mux1* outputs C(0) which is being used and *mux2* outputs G(1) which is being used. M gets its second value, while L gets its first value. W_{e2} , K_2 are the third input values.

The state of *mux1* and *mux2* remains unchanged. With *j* = 3, A–D will get their first new values and E–H their second new values. After the last couple, W_{e63} , K_{63} , has been input, two more clock cycles are needed. The first for generating the final values of E–H, the second for the final values of A–D. During this last clock cycle E–H must not be clocked.

The critical path for the present circuit is:

$$Maj \Rightarrow faa4 \Rightarrow faa5 \Rightarrow cla3$$

or equivalently: $\text{delay}(\text{CLA}) + 3 * \text{delay}(\text{FAA})$.

3.3. A scheme using both delay balancing and pipelining

The new scheme is shown in Fig. 6. We have combined the concepts of pipelining and delay balancing, and we can note that the path leading to A', A'' (α) and to E (ϵ) are split after the common input stage fed by W_{ej} , K_j and H (or G). Two double registers L_1 and L_2 are placed in order to obtain

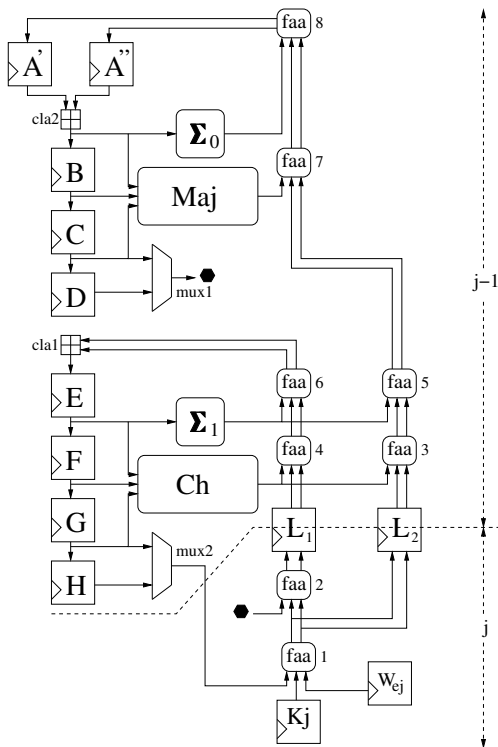


Figure 6. Scheme using delay balancing and pipelining

two pipeline sections: note that (A', A'')–D and E–H belong to the same pipe section and they are always synchronized. This synchronization is achieved in only two clock cycles (as opposed to three in the previous scheme). This is shown in the following sequence of events in the first three clock cycles:

- $j = 0$. A'–H are loaded with the initial Hash values: A(0), B(0), ..., H(0), while A'' is cleared; *mux1* outputs D(0) and *mux2* outputs H(0), which are both used. The registers L₁ and L₂ are cleared. W_{e0} , K_0 are the first input values.
- $j = 1$. (A', A'')–H remain unchanged; *mux1* outputs C(0) and *mux2* outputs G(0), which are both used. L₁ and L₂ get their first new values. W_{e1} , K_1 are the second input values.
- $j = 2$. (A', A'')–H get their first new values; *mux1* outputs C(1) which is being used and *mux2* outputs G(1) which is being used. L₁ and L₂ get their second new values. W_{e2} , K_2 are the third input values.

The operation will continue until $j = 63$. A last clock cycle will be needed to obtain the final hash in (A', A'')–H. The value for the first 32-bit word will be obtained from the output of *cla2*, fed by A', A''.

The longest paths can be seen to be:

$$Ch \Rightarrow faa4 \Rightarrow faa6 \Rightarrow cla1$$

and

$$Ch \Rightarrow faa3 \Rightarrow faa5 \Rightarrow faa7 \Rightarrow faa8$$

Since a CLA is at least three times slower than a FAA, the first path is the critical one, giving a total latency of delay(CLA) + 3*delay(FAA). We must also note that, since Ch contains two-inputs XORs while Maj contains three-inputs XORs, the critical path is slightly shorter than that of Fig. 5.

To better clarify the reason why in the preceding three schemes we had to use two multiplexers, we point out that a straightforward application of pipelining is not possible in this specific case. In fact, because of the circular nature of the circuit, the flow of data related to the α and ϵ functions has a direction opposite to the direction of the memory section composed by the shift registers A–D and E–H. By placing the pipe sections border between D and E we should place there a *negative* delay element, while in the corresponding points in the purely combinational network we will place normal positive delay elements. A negative delay should give the *future* value of D. This would in general be impossible, but in our case the future value of D is simply the actual value of C. The same situation exists for the future value of H in the connection between the pipe section j and the following section $j-1$.

4. Synthesis results

The circuits described in the preceding sections have been coded in VHDL using Mentor HDL Designer Pro, fully simulated and synthesized using Synopsys Design Compiler on a Sun Solaris platform. The target technology library is the aforementioned STMicroelectronics HCMOS8 library, featuring 0.18 μ m process and 1.8V core voltage. The area and timing results are given in Table 2. Area figures are expressed in square micron units and include the Expander circuit; the latency column expresses the number of clock cycles that are needed to complete the hashing of one 512 bit message block (including the accumulation of the intermediate hash).

It is clear from the table that the proposed implementations trade area for speed; the last proposed circuit is capable of running at 819MHz. We can instantiate four such circuits in an ASIC hardware accelerator (achievable from the area point of view): in this way we could take better advantage of the input bandwidth, because each circuit fetches data from the bus only during the first 16 clock cycles ($\frac{16}{64}$, equal to one fourth of the time). In this way we could reach an ideal peak throughput of about 26 Gbits/s.

Circuit	Clock latency	Area	ns/hash
Fig. 2	66×1.47 ns	140644	97.02
Fig. 4	66×1.30 ns	143814	85.8
Fig. 5	68×1.24 ns	159027	84.32
Fig. 6	67×1.22 ns	164856	81.74

Table 2. Comparison of the various implementations

5. Conclusions

In this paper we presented a high speed ASIC implementation of the SHA-256 hash algorithm. Two methods for speeding up the calculation have been introduced and combined, namely delay balancing and pipelining. In particular, the critical path inside the Compressor block has been reduced from delay(CLA) + 5*delay(FAA) in Fig. 2 to delay(CLA) + 3*delay(FAA) in Fig. 6, thus equal to critical path of the Expander. This leads to a possible clock frequency of 819MHz in a 0.18 μ m technology process.

References

- [1] S. Chakrabarti. Design of a universal hashing block for ipsec accelerators. Master's thesis, ALaRI - USI, July 2003.
- [2] L. Dadda. Some schemes for parallel multipliers. *Alta Frequenza*, 34(5):349–356, 1965.
- [3] L. Dadda. On parallel digital multipliers. *Alta Frequenza*, 45(10):574–580, 1976.
- [4] T. Grembowski, R. Lien, K. Gaj, N. Nguyen, P. Bellows, J. Flidr, T. Lehman, and B. Schott. Comparative analysis of the hardware implementations of hash functions sha-1 and sha-512. *Proceedings of the 5th International Conference on Information Security*, pages 75–89, October 2002.
- [5] Y. K. Kang, D. W. Kim, T. W. Kwon, and J. R. Choi. An efficient implementation of hash function processor for ipsec. *Proceedings of the 3rd Asia-Pacific Conference on ASICs*, August 2002.
- [6] NIST. Announcing the secure hash standard. *Federal Information Processing Standards Publication 180-2*, August 2002.
- [7] N. Sklavos and O. Koufopavlou. On the hardware implementations of the sha-2(256,384,512) hash functions. *Proceedings of the IEEE International Symposium on Circuits and Systems ISCAS 2003*, May 2003.
- [8] K. Ting, S. Yuen, K. Lee, and P. Leong. An fpga based sha-256 processor. *Proceedings of the FPL 2002 Conference*, pages 577–585, September 2002.

A. Appendix: the SHA-256 algorithm

Symbols and operators:

- \boxplus = modulo 2^{32} or 2^{64} addition
- \oplus = bitwise XOR
- \wedge = bitwise AND
- $\neg x$ = bitwise complement of x
- $\circlearrowright_n(x)$ = right rotation of x by n bits
- $\rightarrow_n(x)$ = right shift of x by n bits
- $Ch(x, y, z) = (x \wedge y) \oplus (\neg x \wedge z)$
- $Maj(x, y, z) = (x \wedge y) \oplus (x \wedge z) \oplus (y \wedge z)$
- $\Sigma_0(x) = \circlearrowright_2(x) \oplus \circlearrowright_{13}(x) \oplus \circlearrowright_{22}(x)$
- $\Sigma_1(x) = \circlearrowright_6(x) \oplus \circlearrowright_{11}(x) \oplus \circlearrowright_{25}(x)$
- $\sigma_0(x) = \circlearrowright_7(x) \oplus \circlearrowright_{18}(x) \oplus \rightarrow_3(x)$
- $\sigma_1(x) = \circlearrowright_{17}(x) \oplus \circlearrowright_{19}(x) \oplus \rightarrow_{10}(x)$

The message to be hashed M is first padded and then divided in N 512-bit blocks $M^{(1)}, M^{(2)}, \dots, M^{(N)}$. Each block i is made up by 16 32-bit words $M_0^{(i)}, M_1^{(i)}, \dots, M_{15}^{(i)}$. The hash computation is accomplished by carrying out the following steps, for $i=1$ to N :

- The registers A–H are initialized with the $(i-1)^{th}$ intermediate hash value:
 $A \leftarrow H_1^{(i-1)}, B \leftarrow H_2^{(i-1)}, \dots, H \leftarrow H_8^{(i-1)}$.
- For $j=0$ to 63 the compression function is applied:
 $T_1 \leftarrow H \boxplus \Sigma_1(E) \boxplus Ch(E, F, G) \boxplus K_j \boxplus W_j$
 $T_2 \leftarrow \Sigma_0(A) \boxplus Maj(A, B, C)$
 $H \leftarrow G$
 $G \leftarrow F$
 $F \leftarrow E$
 $E \leftarrow D \boxplus T_1$
 $D \leftarrow C$
 $C \leftarrow B$
 $B \leftarrow A$
 $A \leftarrow T_1 \boxplus T_2$
- The i^{th} intermediate hash value is calculated:
 $H_1^{(i)} \leftarrow A \boxplus H_1^{(i-1)}$,
 $H_2^{(i)} \leftarrow B \boxplus H_2^{(i-1)}$,
 \dots ,
 $H_8^{(i)} \leftarrow H \boxplus H_8^{(i-1)}$.

At the end of the process, $H^{(N)} = (H_1^{(N)}, H_2^{(N)}, \dots, H_8^{(N)})$ is the hash of the message M . The values W_j are obtained from the message schedule (the Expander):

- $W_j = M_j^{(i)}$ for $j = 0, 1, \dots, 15$
- $W_j = \sigma_1(W_{j-2}) \boxplus W_{j-7} \boxplus \sigma_0(W_{j-15}) \boxplus W_{j-16}$ for $j = 16, \dots, 63$

The values of the initial hash $H^{(0)}$ and of the constants K_j can be found in [6], along with the instructions on how to pad the message.