7

Reconfigurable Hardware Implementation of Hash Functions

This Chapter has two main purposes. The first purpose is to introduce readers to how hash functions work. The second purpose is to study key aspects of hardware implementations of hash functions. To achieve those goals, we selected MD5 as the most studied and widely used hash algorithm. A step-by-step description of MD5 has been provided which we hope will be useful for understanding the mathematical and logical operations involved in it. The study and analysis of MD5 will be utilized as a base for explaining the most recent SHA2 family of hash algorithms.

We start this Chapter given a brief introduction to hash algorithms in Section 7.1. A survey of some famous hash algorithms is presented in Section 7.2. Then we provide a detailed discussion of the MD5 algorithm in Sec. 7.3. All MD5 steps are explained by means of an illustrative example which is explained at a bit level. In Section 7.4, we describe the SHA2 family of hash algorithms and some tips are provided with respect to their hardware implementation. In Section 7.5 design strategies to achieve efficient hash algorithms when implemented on reconfigurable devices are discussed. Section 7.6 presents a review of recent hash function hardware implementations. Finally, in Section 7.7 concluding remarks are drawn.

7.1 Introduction

As it was explained in Chapter 2, a Hash function H is a computationally efficient function that maps fixed binary chains of arbitrary length $\{0,1\}^*$ to bit sequences $H(B)$ of fixed length. $H(M)$ is the hash value, hash code or digest of M [110].

In words, let M be a message of an arbitrary length. A *hash function* operates on M and returns a fixed-length value, h, as shown in Fig. 7.1. The value h is commonly called *hash code*. It is also referred to as a message digest or hash value. The main application of hash functions lies on producing fingerprint of a file, message or other blocks of data.

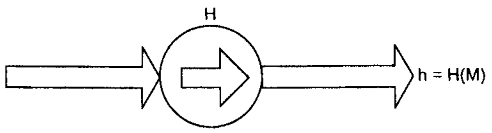

Fig. 7.1. Hash Function

Hash functions do not use a particular key, but instead, it is a highly non linear function of all message bits. The code changes with the change of any bit or bits in the input message and thus it provides error detection capabilities.

In practice, modern hash functions are specifically designed for having a short bit-length hash code h (usually from around 128 bits up to 512 bits). This characteristic is especially attractive for the application of hash functions in virtually every digital signature algorithm. Therefore, rather than attempting to sign the whole message (which by definition has arbitrary length), it becomes more practical to sign the hash code of the message as it was depicted in the basic digital signature/verification scheme shown in Figure 2.6.

As a way of illustration, let us suppose that Ana received \$500 from Bill, and that afterwards, she proceeded signing the hash code $h1$ of the message $M1$ as shown below,

M1 = Ana received \$500 from Bill

h1 = H(M1) = 89CB0C238A3C7A78D0DD7063C4153B65

Bill can never claim that Ana received \$5000 as the hash code $h2$ of message $M2$ using the same hash function vastly differs,

M2 = Ana received \$5000 from Bob.

h2=H(M2)=CCD40B907C543D96FDB7203979E55E8B

Alternatively, Bill may try to find another message M3 whose hash value corresponds to the hash value of message M1, and then claim that Ana actually signed message M3, not M1.

If we can find any two messages producing the same message digest, we say that we have found a *collision*. *Collision* is a not desired characteristic of hash functions but at the same time is unavoidable. All that one can hope is that no matter how determined an adversary may be, it should result computational unfeasible for him/her to find collisions. Therefore, a hash function H is said to be strong enough against collision and thus useful for message authentication, if it has the following properties [342, 246],

- H applies to any block of data.
- H returns a fixed-length output.
- For any given value x, $H(x)$ is relatively easy to compute. That feature makes hash function implementations more practical in both software and hardware platforms (Fig. 7.2a).

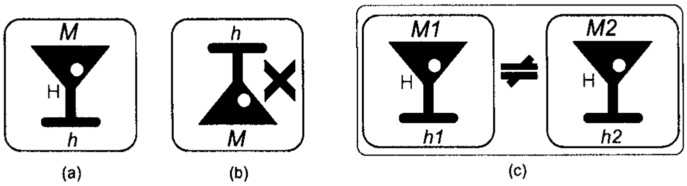

Fig. 7.2. Requirements of a Hash Function

- Given x, it is easy to compute $H(x)$. Given h, it is computationally infeasible to find x such that $H(x) = h$. That is sometimes referred to as *one way* property of hash functions (Fig. 7.2b).
- For any given block x, it is computationally infeasible to find y ($y \neq x$), with $H(y) = H(x)$. This is sometimes referred to as *weak collision resistance.*
- To find a pair (x, y) such that $H(x) = H(y)$, is computationally infeasible. This is sometimes referred to as *strong collision resistance* (Fig. 7.2c).

7.2 Some Famous Hash Functions

The overall structure of a typical hash function is shown in Fig. 7.3.

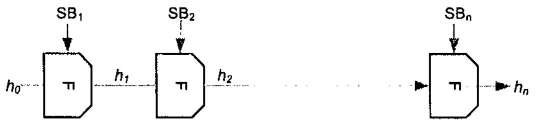

Fig. 7.3. Basic Structure of a Hash Function

The structure was first proposed by Merkle [233, 234] and then followed by most hash function designs in use today including MD5, SHA-1 and RIPEMD-160 [342].

It is apparent from Fig. 7.3 that a typical hash function is iterative in nature. That is, it partitions (*hashes*) a given input message to L sub blocks SBs of some fixed length m bits and operates sequentially on each SB. Those message blocks shorter in length than m are padded as necessary with zeroes.

Table 7.1. Some Known Hash Functions

Name	Author(s)	Year	Block Size	Digest Size
AR	ISO [151]	1992		
Boognish	Daemen[58]	1992	32	up to 160
Cellhash	Daemen, Govaerts, Vandewalle [59]	1991	32	up to 256
FFT-Hash I	Schnorr [318]	1991	128	128
GOST R 34.11-94	Government Committee of Russia for Standards [257]	1990	256	256
FFT-Hash II	Schnorr [319]	1992	128	128
HAVAL	Zheng, Pieprzyk, Seberry [402]	1994	1024	128, 160, 192, 224, 256
MAA	ISO [150]	1988	32	32
MD2	Rivest [162]	1989	512	128
MD4	Rivest [288]	1990	512	128
MD5	Rivest [289]	1992	512	128
N-Hash	Miyaguchi, Ohta, Iwata [237]	1990	128	128
PANAMA	Daemen, Clapp [56]	1998	256	unlimited
Parallel FFT-Hash	Schnorr, Vaudenay [320]	1993	128	128
RIPEMD	The RIPE Consortium [287]	1990	512	128
RIPEMD-128	Dobbertin, Bosselaers, Preneel [70]	1996	512	128
RIPEMD-160	Dobbertin, Bosselaers, Preneel [70]	1996	512	160
SHA-0	NIST/NSA [61]	1991	512	160
SHA-1	NIST/NSA [255]	1993	512	160
SHA-224	NIST/NSA [255]	2004	512	224
SHA-256	NIST/NSA [255]	2000	512	256
SHA-384	NIST/NSA [255]	2000	1024	384
SHA-512	NIST/NSA [255]	2000	1024	512
SMASH	Knudsen [177]	2005	256	256
Snefru	Merkle [235]	1990	512-m	m = 128, 256
StepRightUp	Daemen [55]	1995	256	256
Subhash	Daemen [57]	1992	32	up to 256
Tiger	Anderson, Biham [8]	1996	512	192
Whirlpool	Barreto, Rijmen [286]	2000	512	512

The heart of a hash algorithm is the so-called *compression function* F. A repeated use of function F is made by the hash algorithm. F takes two inputs: an m-bit input block message and; an n-bit input from previous step, called hash h of that message block. The output is an n-bit hash h, namely [317],

$$h_j = F(Sb_j, h_{j-1}) \tag{7.1}$$

For j=1, 2, . . . , L, where L is the total number of SB message blocks. For j = 1, the function F takes the first sub block SB_1 and h_0, where h_0 is a fixed value provided by the algorithm. For h_n, (i.e. $j = n$), the two inputs are SB_n and h_{n-1}, h_n is the hash value of the entire message.

The term compression comes from the fact that the hash output has a much shorter bit-length n than the original input message bit-length m. Although it has not been formally proved, some authors consider that the security of a hash function strongly depends upon the security of its compression function [234, 62, 245]. Indeed, if the compression function is strongly collision resistant, then hashing a message using that method is also secure. Modern hash functions strive for improving the internal logic of their compression functions. At the same time, extensive research has been carried out on the issue of how many repetitions of the compression function are essential for obtaining an acceptable security and how those repetitions could be sequenced.

Table 7.1 features a list of known hash functions prepared by [17]. Detailed discussions about the design of most of those hash functions can be found in [165, 275, 234, 19, 276, 277, 276, 278, 347, 348, 360, 28, 119, 119, 138].

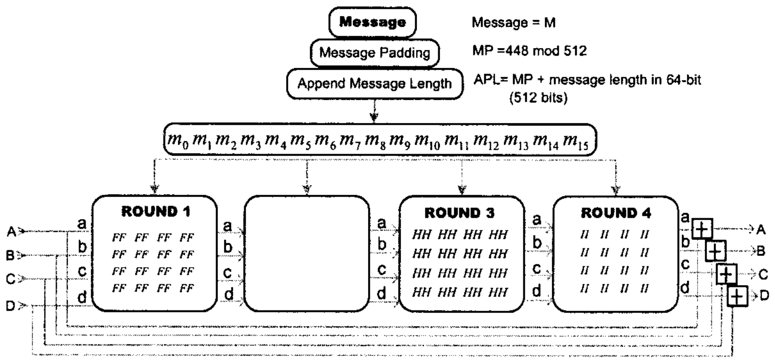
Message
Message Padding
Append Message Length
Message = M
MP =448 mod 512
APL= MP + message length in 64-bit (512 bits)
m_0 m_1 m_2 m_3 m_4 m_5 m_6 m_7 m_8 m_9 m_{10} m_{11} m_{12} m_{13} m_{14} m_{15}
ROUND 1
FF FF FF FF
ROUND 3
HH HH HH HH
ROUND 4
II II II II
a
b
c
d
A
B
C
D

Fig. 7.4. MD5

7.3 MD5

The series of Message Digest (MD) hash algorithms is due to Rivest[289]. The original message digest algorithm was simply called MD. MD was quickly followed by MD2 [162]. Nevertheless, MD2 was soon found to be quite weak. Rivest then started working on MD3, which however was never released. MD4 [288] was the next family member. Soon MD4 was also found to be imperfect, but it provided the theoretical foundations for its successors MD5 (designed in 1992) and also for SHA-0 [61] and RIPEMD [287], from other authors. Then, in 2004, the never ending battle between hash function designers and crypto analysts had yet another episode, when several advances for finding collisions on MD5 were announced in [24, 159].

Short after that, Wang et al. without revealing their method, presented on the rump session of [98] evidence of MD5 colliding messages [370]. Wang et al. method was later published in [372]. Before that happened though, several experimental results were presented in [174], showing for the first time how MD5 could be break. Recently, it has been proved that collisions on MD5 can be found (under certain conditions) within a minute using a standard laptop [175].

Operating on 512-bit input blocks, MD5 produces 128-bit message digests from input messages of arbitrary length. For longer messages, a partition into sub blocks is performed. The algorithm then operates iteratively on all message sub-blocks as shown in Fig. 7.4. In the following Subsection, MD5 steps for hashing a message are described in detail.

7.3.1 Message Preprocessing

First, original message is preprocessed. The message is padded such that its length (in bits) is congruent to 448 mod 512. Messages shorter than 448 bits are padded with the first bit set to '1' and all the rest set to zero. The remaining 64 bits for completing a block of 512 bits are reserved for appending message length. For instance, a message with 200-bit length would require a padding of 228 bits. The padding would comprise a single '1' at the most significant position followed by 227 zeroes. The last 64 bits are all zeroes except for the last byte which is "11001000" denoting message length of 200. As a way of illustration, we show below how a sub block of 512-bit is obtained from an input message. Let our input message M be,

"MD5 was proposed by Ron Rivest in 1992."

The ASCII representation of the message M (39 characters) is shown in Table 7.2.

Table 7.2. Bit Representation of the Message M

01001101	01000100	00110101	00100000	01110111	01100001	01110011	00100000
01110000	01110010	01101111	01110000	01101111	01110011	01100101	01100100
00100000	01100010	01111001	00100000	01010010	01101001	01110110	01100101
01110011	01110100	00100000	01101001	01101110	00100000	00110001	00111001
00111001	00110010	00101110					

The first step consists on padding the Message M in order to complete a block of 512 bits as shown in Table 7.3. Notice the location of the padding start bit (i.e. bit '1') and the message length (given in a 64-bit representation) appended into the last 64 bits (shaded). As it was explained above, the padding process assures that the block message length will always be an exact multiple of 512. Thereafter the main loop starts. A message parsing is required for this loop. This is accomplished by dividing the 512-bit input message block into sixteen 32 bit words.

Table 7.3. Padded Message (M)

01001101	01000100	00110101	00100000	01110111	01100001	01110011	00100000
01110000	01110010	01101111	01110000	01101111	01110011	01100101	01100100
00100000	01100010	01111001	00100000	01010010	01101001	01110110	01100101
01110011	01110100	00100000	01101001	01101110	00100000	00110001	00111001
00111001	00110010	00101110	10000000	00000000	00000000	00000000	00000000
00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000
00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000
00000000	00000000	00000000	00000000	00000000	00000000	00000001	00011000

In the case of hardware implementations, designers can use various options for message preprocessing. One of the possible approaches is to use sixteen 32 bit shift registers which are initialized with zeroes except for the first one which has its first bit set to '1'. All the 16 registers are cascaded in such a way that the output of one is placed as the input of the next register.

Thus, whenever a message is read, all message bits are sequentially transferred to shift registers. The start bit '1' of the first shift register is now the end bit of the message as shown in Fig. 7.5. Since there is no need to cascade final register (SR15) with the other registers it can be reserved for appending the message length. That register arrangement also completes message parsing as all 16 registers contain 32-bit words.

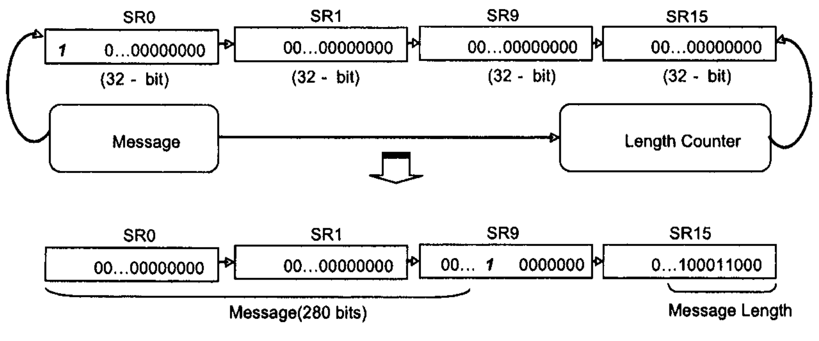

Fig. 7.5. Message Block = 32 × 16 =512 Bits

Rivest selected a *little-endian architecture* for interpreting a message as a sequence of 32-bit words. A little endian architecture stores the least significant byte of a word into the lowest byte address. This design decision was taken due to Rivest observation that several processor architectures with little endian format offer faster processing [342]. This way, the first block message is converted into sixteen 32-bit words, which are then written into hex little endian format as shown in Table 7.4.

Table 7.4. Message in Little Endian Format

Message in Hex	Message little endian format
0x4d443520	0x2035444d
0x77617320	0x20736177
0x70726f70	0x706f7270
0x6f736564	0x6465736f
0x20967920	0x20796220
0x526f6e20	0x206e6f52
0x52697665	0x65766952
0x69207473	0x69207473
0x6e203139	0x3931206e
0x39322e80	0x802e3239
0x00000000	0x00000000
0x00000000	0x00000000
0x00000000	0x00000000
0x00000000	0x00000000
0x00000000,0x00000138	0x00000138,0x00000000

Appending bits to message blocks according to the Little endian format is intended for 32-bit word rather than one byte words. Therefore, the 64 bits that are reserved for keeping the message length are divided into two 32-bit words. By applying said convention, the lower order 32-bit word is appended first as shown in Table 7.4 (observe the last two 32-bit words).

7.3.2 MD Buffer Initialization

As it has been already mentioned, internally MD5 operates on two inputs: the input message block and the output hash from the previous step. In the first step, the initial hash values are constants provided by the algorithm. The initial values for MD5 are provided into four 32-bit words. A four-word buffer (a, b, c, d) is used to store those values which are then replaced by the output hash values after each step. MD5 a, b, c, d four words, are also referred to as *chain variables.* The initial values for the MD5 chain variables are shown in Table 7.5.

Table 7.5. Initial Hash Values in Little Endian Format

Normal Values	Little endian format
a = 0x01234567	a = 0x67452301
b = 0x89abcdef	b = 0xefcdab89
c = 0xfedcba98	c = 0x98badcfe
d = 0x76543210	d = 0x10325476

7.3.3 Main Loop

The Main loop is composed of four rounds. Each round has as a 512-bit message block as an input. As it was mentioned, message blocks are grouped into sixteen 32-bit words. The second input comes in the form of chain variables which are also grouped as four words of 32-bit each (totaling 128 bits). All the four rounds use an auxiliary function, which takes three 32-bit inputs producing a single 32-bit output. Table 7.6 presents the four non-linear functions F, G, H, and I, that are utilized in rounds 1 to 4.

Table 7.6. Auxiliary Functions for Four MD5 Rounds

F(A,B,C)	= (A AND B) OR ((NOT A) AND C)
G(A,B,C)	= (A AND C) OR (B AND (NOT C))
H(A,B,C)	= (A XOR B XOR C)
I(A,B,C)	= (B XOR (A OR (NOT C)))

All the four non-linear functions are simple and can be easily constructed in reconfigurable hardware. The architecture of those four functions maps well to those reconfigurable devices having a 4-bit input/1-bit output Look Up Tables (LUTs) as a basic unit. On such devices, all the four functions occupy a single LUT, thus using a total of 4 LUTs for one bit manipulation as shown in Fig. 7.6.

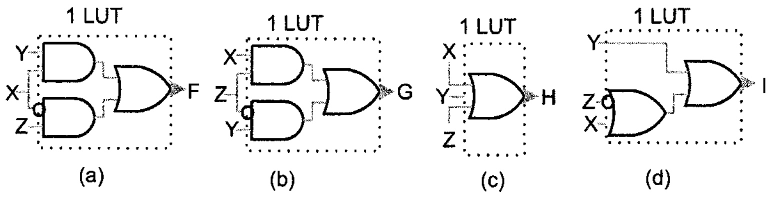

Fig. 7.6. Auxiliary Functions in Reconfigurable Hardware (a) F(X,Y,Z) (b) G(X,Y,Z) (c) H(X,Y,Z) (d) I(X,Y,Z)

Let $\lll S$ denote a left circular shift by S bits and let m_i represent the ith sub-block (0 to 15) of the message. Provided that there is a constant K_j for the jth state of a round, the four operations corresponding to four MD5 rounds are shown in Table 7.7.

Table 7.7. Four Operations Associated to Four MD5 Rounds

$$\mathrm{FF}(a,b,c,d,\ m_i,\ S,\ K_j) \Rightarrow a = b + ((a + F(b,c,d) + m_i + K_j) \lll S)$$
$$\mathrm{GG}(a,b,c,d,\ m_i,\ S,\ K_j) \Rightarrow a = b + ((a + G(b,c,d) + m_i + K_j) \lll S)$$
$$\mathrm{HH}(a,b,c,d,\ m_i,\ S,\ K_j) \Rightarrow a = b + ((a + H(b,c,d) + m_i + K_j) \lll S)$$
$$\mathrm{II}(a,b,c,d,\ m_i,\ S,\ K_j) \Rightarrow a = b + ((a + I(b,c,d) + m_i + K_j) \lll S)$$

The architecture of a single MD5 operation can be optimized for reconfigurable devices by re-ordering some steps as shown in Fig. 7.7.

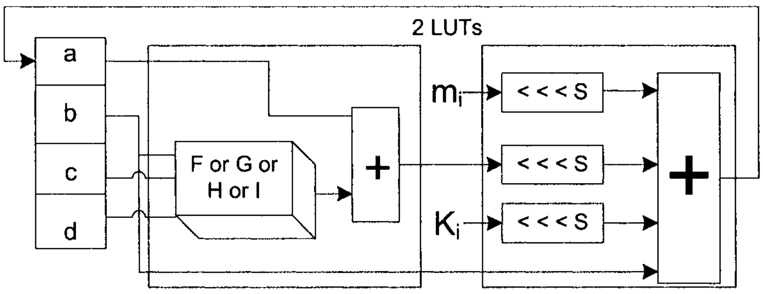

Fig. 7.7. One MD5 Operation

Two changes are introduced. First, summation of word a is appended with the manipulation of the non-linear function, this occupies a single LUT. Similarly, instead of a single shift operation by S bits, a total of three shift operations have been introduced. That does not cost other logic resources but only the routing resources of the target reconfigurable device.

There are a total of 64 steps in the four MD5 rounds. The output of each round for our example message is presented in Table 7.8, Table 7.9, Table 7.10, and Table 7.11 for round 1, round 2, round3, and round 4, respectively. The constant values K_i can be computed by taking the integer part of $2^{32} \times abs(sin(i))$, where i is in radians.

7.3.4 Final Transformation

The last step consists on adding the initial and final hash values. Here addition is a simple integer addition modulo 2^{32} and not an 'XOR' operation. The

Table 7.8. Round 1

Function	Output
FF (a, b, c, d, m_0, 7, 0xd76aa478)	a = 0xbfc20e04
FF (d, a, b, c, m_1, 12, 0xe8c7b756)	d = 0x2445ea9a
FF (c, d, a, b, m_2, 17, 0x242070db)	c = 0xbada24bf
FF (b, c, d, a, m_3, 22, 0xc1bdceee)	b = 0xdae8f105
FF (a, b, c, d, m_4, 7, 0xf57c0faf)	a = 0xd3e2a4f
FF (d, a, b, c, m_5, 12, 0x4787c62a)	d = 0x618adec1
FF (c, d, a, b, m_6, 17, 0xa8304613)	c = 0x605da696
FF (b, c, d, a, m_7, 22, 0xfd469501)	b = 0xb10d4538
FF (a, b, c, d, m_8, 7, 0x698098d8)	a = 0xf0ce7848
FF (d, a, b, c, m_9, 12, 0x8b44f7af)	d = 0xadc2ea19
FF (c, d, a, b, m_{10}, 17, 0xffff5bb1)	c = 0x8ca10c71
FF (b, c, d, a, m_{11}, 22, 0x895cd7be)	b = 0xd06eda96
FF (a, b, c, d, m_{12}, 7, 0x6b901122)	a = 0xcfc79c1a
FF (d, a, b, c, m_{13}, 12, 0xfd987193)	d = 0xef0992d6
FF (c, d, a, b, m_{14}, 17, 0xa679438e)	c = 0x419bb7da
FF (b, c, d, a, m_{15}, 22, 0x49b40821)	b = 0xa41613f9

Table 7.9. Round 2

Function	Output
GG (a, b, c, d, m_1, 5, 0xf61e2562)	a = 0x01816d6a
GG (d, a, b, c, m_6, 9, 0xc040b340)	d = 0x8d2b14de
GG (c, d, a, b, m_{11}, 14, 0x265e5a51)	c = 0xf0ec903d
GG (b, c, d, a, m_0, 20, 0xe9b6c7aa)	b = 0xfbb03b00
GG (a, b, c, d, m_5, 5, 0x0d62f105d)	a = 0x3c1fe25e
GG (d, a, b, c, m_{10}, 9, 0x02441453)	d = 0x53c87df3
GG (c, d, a, b, m_{15}, 14, 0xd8a1e681)	c = 0xefcf863a
GG (b, c, d, a, m_4, 20, 0xe7d3fbc8)	b = 0x7a06c30d
GG (a, b, c, d, m_9, 5, 0x21e1cde6)	a = 0x00fb73e8
GG (d, a, b, c, m_{14}, 9, 0xc33707d6)	d = 0x968fd037
GG (c, d, a, b, m_3, 14, 0xf4d50d87)	c = 0x14952739
GG (b, c, d, a, m_8, 20, 0x455a14ed)	b = 0xcf0e19b2
GG (a, b, c, d, m_{13}, 5, 0xa9e3e905)	a = 0xeec09e98
GG (d, a, b, c, m_2, 9, 0xfcefa3f8)	d = 0xe0cb123e
GG (c, d, a, b, m_7, 14, 0x676f02d9)	c = 0xadfb03b9
GG (b, c, d, a, m_{12}, 20, 0x8d2a4c8a)	b = 0x3d9b93ef

Table 7.10. Round 3

Function	Output
HH (a, b, c, d, m_5, 4, 0xfffa3942)	a = 0x3ae82d36
HH (d, a, b, c, m_8, 11, 0x8771f681)	d = 0xf21c9795
HH (c, d, a, b, m_{11}, 16, 0x6d9d6122)	c = 0x8043a89c
HH (b, c, d, a, m_{14}, 23, 0xfde5380c)	b = 0x3985c48b
HH (a, b, c, d, m_1, 4, 0xa4beea44)	a = 0xf8dd0bbf
HH (d, a, b, c, m_4, 11, 0x4bdecfa9)	d = 0x7a6540bb
HH (c, d, a, b, m_7, 6, 0xf6bb4b60)	c = 0x7263dc17
HH (b, c, d, a, m_{10}, 23, 0xbebfbc70)	b = 0x79d86ca3
HH (a, b, c, d, m_{13}, 4, 0x289b7ec6)	a = 0xaf5015ec
HH (d, a, b, c, m_0, 11, 0xeaa127fa)	d = 0xe9e2e73d
HH (c, d, a, b, m_3, 16, 0xd4ef3085)	c = 0x860d260
HH (b, c, d, a, m_6, 23, 0x4881d05)	b = 0xddfa26e9
HH (a, b, c, d, m_9, 4, 0xd9d4d039)	a = 0x3aace80d
HH (d, a, b, c, m_{12}, 11, 0xe6db99e5)	d = 0xdf9a1e0c
HH (c, d, a, b, m_{15}, 16, 0x1fa27cf8)	c = 0xffda7edc
HH (b, c, d, a, m_2, 23, 0xc4ac5665)	b = 0x4d718018

Table 7.11. Round 4

Function	Output
II (a, b, c, d, m_0, 6, 0xf4292244)	a = 0xbc2cf190
II (d, a, b, c, m_7, 10, 0x432aff97)	d = 0xc43bf785
II (c, d, a, b, m_{14}, 15, 0xab9423a7)	c = 0x9d557285
II (b, c, d, a, m_5, 21, 0xfc93a039)	b = 0xbf063e88
II (a, b, c, d, m_{12}, 6, 0x655b59c3)	a = 0xc5ec3319
II (d, a, b, c, m_3, 10, 0x8f0ccc92)	d = 0x20d2175b
II (c, d, a, b, m_{10}, 15, 0xffeff47d)	c = 0xc6863889
II (b, c, d, a, m_1, 21, 0x85845dd1)	b = 0xf70ea106
II (a, b, c, d, m_8, 6, 0x6fa87e4f)	a = 0x12f76270
II (d, a, b, c, m_{15}, 10, 0xfe2ce6e0)	d = 0xd40a121f
II (c, d, a, b, m_6, 15, 0xa3014314)	c = 0xe4c960a4
II (b, c, d, a, m_{13}, 21, 0x4e0811a1)	b = 0x2fb93bf8
II (a, b, c, d, m_4, 6, 0xf7537e82)	a = 0xadf1d7b5
II (d, a, b, c, m_{11}, 10, 0xbd3af235)	d = 0xfd93443b
II (c, d, a, b, m_2, 15, 0x2ad7d2bb)	c = 0x5a402c56
II (b, c, d, a, m_9, 21, 0xeb86d391)	b = 0x9f2895cb

resultant four words a, b, c, and d would be in little-endian format. They need to be converted back to its original format. Finally, four words a, b, c, and d are concatenated to give the 128-bit hash of the given message as shown in Table 7.12.

Table 7.12. Final Transformation

Initial Hash Values	Round Output	Final Transformation	Conversion from Little Endian
a = 0x67452301	b = 0xefcdab89	c = 0x98badcfe	d = 0x10325476
a = 0xadf1d7b5	b = 0x9f2895cb	c = 0x5a402c56	d = 0xfd93443b
a = 0x1536fab6	b = 0x8ef64154	c = 0xf2fb0954	d = 0x0d508c19
a = 0xb6fa3615	b = 0x5441f68e	c = 0x5409fbf2	d = 0xb198c50d

Final Hash = b6fa36155441f68e5409fbf2b198c50d

7.4 SHA-1, SHA-256, SHA-384 and SHA-512

The FIPS 180-2 [255] supersedes FIPS 180-1 [95]. It includes four secure hash algorithms SHA-1, SHA-224, SHA-384 and SHA-512. SHA-1 is identical to SHA-1 specified in FIPS 180-1[1].

Some notational changes have been introduced to make it consistent with the other three algorithms. All four algorithms are one way iterative hash functions. They differ in terms of block and word size. They also differ in the size of the message digest, which redounds in different levels of security. Table 7.13 compares basic specifications of the four secure hash algorithms.

Table 7.13. Comparing Specifications for Four Hash Algorithms

Algorithm	Message Size (bits)	Block Size (bits)	Word Size (bits)	Message Digest (bits)	Security (bits)
SHA-1	$< 2^{64}$	512	32	160	80
SHA-256	$< 2^{64}$	512	32	256	128
SHA-384	$< 2^{128}$	1024	64	384	192
SHA-512	$< 2^{128}$	1024	64	512	256

[1] Just as it happened with MD5, the SHA family of hash algorithms has been successfully attacked in several recent papers [371, 107].

7.4.1 Message Preprocessing

Preprocessing is always done before hash computation begins. Preprocessing comprises three main steps,

Step 1: Padding the message
Step 2: Parsing the padded message
Step 3: Setting the initial hash values

The hash computation for SHA-1 and SHA-256 requires 512-bit block. A 1024-bit input block is processed by SHA-384 and SHA-512 hash computation. Preprocessing for both categories is discussed separately.

SHA-1 and SHA-256

Step 1: Padding the Message

Let l be the length of the message M in bits. Append bit '1' to
the end of the message followed by k zeroes such that the length of the resulting block is 64 bits short of 512 bits, i.e.,

$$\text{Result} = M + 1 + k = 448 \bmod 512.$$

The remaining 64 bits are reserved for adding the message length l in its binary representation. As an example, the message 'try' has an ASCII representation of 24 bits (8 × 3). Therefore, it requires 423 more bits to be padded at the end of the message in addition to the leading bit '1' in order to complete a block of 448 bits. The message length $l = 24$ in its 64-bit Boolean representation is appended at the end, as shown in Fig. 7.8.

$$\underbrace{01110100}_{\text{t}}\ \underbrace{01110010}_{\text{r}}\ \underbrace{01111001}_{\text{y}}\ 1\ \overbrace{00.....00}^{423}\ \ \overbrace{00...0\underbrace{11000}_{l=24}}^{64}$$

Fig. 7.8. Padding Message in SHA-1 and SHA-256

Padding is always made even if the message block is of 448 bits. For a 448-bit message, a single bit '1' is appended at the end followed by 447 zeroes. Thus, in that case, an apparent single block message would be treated as two separated blocks.

Step 2 : Parsing the message

A padded message is parsed to N 512-bit blocks, namely, $M_0, M_1, \ldots, M_N$. Where each M_i block is organized into sixteen 32-bit blocks, namely, M_i^0, M_i^1, $\ldots, M_i^{15}$. Therefore, the first sixteen 32-bit blocks are: $M_0^0, M_0^1, \ldots, M_0^{15}$.

Step 3: Setting the initial hash values

Before beginning the actual hash function computation, initial values must be set. Those values are provided by the algorithm. Table 7.14 and Table 7.15 show in hex format five 32-bit words for SHA-1 and eight 32-bit words for SHA-256, respectively.

Table 7.14. Initial Hash Values for SHA-1

a =	0x67452301
b =	0xefcdab89
c =	0x98badcfe
d =	0x10325476
e =	0xc3d2e1f0

Table 7.15. Initial Hash Values for SHA-256

a =	0x6a09e667
b =	0xbb67ae85
b =	0x3c6ef372
c =	0xa54ff53a
d =	0x510e527f
e =	0x9b05688c
f =	0x1f83d9ab
g =	0x5be0cd19

SHA-384 and SHA-512

Step 1: Padding the message

Padding procedure for SHA-384 and SHA-512 is similar to those of SHA-1 and SHA-256. However, let us recall that both SHA-384 and SHA-512 operate on 1024-bit message blocks, which consequently causes a change in other lengths. Let l be the length of the message M in bits. In this case, after appending a single bit '1' to the end of the message, k zeroes are added such that the length of the resulting block is 120 bits short of 1024 bits,

$$\text{Result} = M + 1 + k = 896 \bmod 1024$$

The remaining 120 bits are reserved for appending the message length l in its binary representation. Once again, let us consider the same example message "try" (24 bits). In this case, 871 more bits are required to be padded at the end of the message in addition to the mandatory leading bit '1' to complete a block of 896 bits. The remaining 120 bits represent the message length as shown in Fig.7.9.

$$\underbrace{01110100}_{t}\ \underbrace{01110010}_{r}\ \underbrace{01111001}_{y}\ 1\ \overbrace{00.....00}^{423}\quad \overbrace{00...0\underbrace{11000}_{l=24}}^{64}$$

Fig. 7.9. Padding Message in SHA-384 and SHA-512

Step 2 : Parsing the message

Padded messages are parsed to N 1024-bit blocks: $M_0, M_1, \ldots, M_N$. Where each M_i comprises thirty-two 32-bit blocks, namely, $M_i^0, M_i^1, \ldots, M_i^{31}$. The first thirty-two 32 blocks are $M_0^1, M_0^2, \ldots, M_0^{31}$, and so on.

Step 3: Setting the initial hash values

The initial values SHA-384 and SHA-512 comprises two sets of eight 64-bit words as shown in Table 7.16 and Table 7.17.

Table 7.16. Initial Hash Values for SHA-384

a	=	0xcbbb9d5dc1059ed8
b	=	0x629a292a367cd507
c	=	0x9159015a3070dd17
d	=	0x152fecd8f70e5939
e	=	0x67332667ffc00b31
f	=	0x8eb44a8768581511
g	=	0xdb0c2e0d64f98fa7
h	=	0x47b5481dbefa4fa4

7.4.2 Functions

The auxiliary functions used in SHA-1 differ to those functions used in SHA-256, SHA-384 and SHA-512. Functions used in SHA-256, SHA-384 and SHA-512 are identical but they operate on different word sizes.

Table 7.17. Initial Hash Values for SHA-512

a	= 0x6a09e667f3bcc908
b	= 0xbb67ae8584caa73b
c	= 0x3c6ef372fe94f82b
d	= 0xa54ff53a5f1d36f1
d	= 0x510e527fade682d1
e	= 0x9b05688c2b3e6c1f
f	= 0x1f83d9abfb41bd6b
g	= 0x5be0cd19137e2179

7.4.3 SHA-1

The function F_t in SHA-1 takes three 32-bit words X, Y, and Z, producing a single 32-bit word output, where the variable t ranges from 0 to 79. It is defined as indicated below.

$$F_t = \begin{cases} Ch(X,Y,Z) = (X \text{ } OR \text{ } Y) \oplus ((NOT \text{ } X) \text{ } OR \text{ } Z) & 0 \leq t \leq 19 \\ Parity(X,Y,Z) = X \oplus Y \oplus Z & 20 \leq t \leq 39 \\ Maj(X,Y,Z) = (X \text{ } OR \text{ } Y) \oplus (X \text{ } OR \text{ } Z) \oplus (Y \text{ } OR \text{ } Z) & 40 \leq t \leq 59 \\ Parity(X,Y,Z) = X \oplus Y \oplus Z & 60 \leq t \leq 79 \end{cases}$$

A reconfigurable hardware architecture for the F_t is illustrated in Fig. 7.10. It is noted that all three, *Ch*, *Parity*, and *Maj*, occupy a single LUT when 1-bit operand is processed.

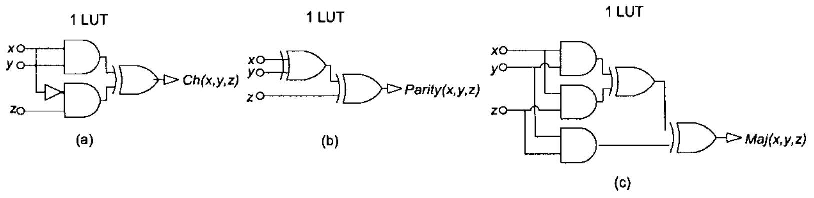

Fig. 7.10. Implementing SHA-1 Auxiliary Functions in Reconfigurable Hardware

SHA-256, SHA-384 and SHA-512

All three, SHA-256, SHA-384 and SHA-512, use six logical functions. Each function operates on three words X, Y, and Z producing a new word of the same size as output. SHA-256 operates on 32-bit long words X, Y and Z. However, both SHA-384 and SHA-512 operates on 64-bit words. The six functions are,

$$\begin{aligned}
Ch(X,Y,Z) &= (X\ OR\ Y) \oplus ((NOT\ X)\ OR\ Z)\\
Maj(X,Y,Z) &= (X\ OR\ Y) \oplus (X\ OR\ Z) \oplus (Y\ OR\ Z)\\
\Sigma_0(X) &= ROTR^2(X) \oplus ROTR^{13}(X) \oplus ROTR^{22}(X)\\
\Sigma_1(X) &= ROTR^6(X) \oplus ROTR^{11}(X) \oplus ROTR^{25}(X)\\
\sigma_0(X) &= ROTR^7(X) \oplus ROTR^{18}(X) \oplus ROTR^3(X)\\
\sigma_1(X) &= ROTR^{17}(X) \oplus ROTR^{19}(X) \oplus ROTR^{10}(X)
\end{aligned}$$

The architectures for $Ch(X,Y,Z)$ and $Maj(X,Y,Z)$ are identical to the architectures presented in Fig. 7.10. The architectures for Σ_0, Σ_1, σ_0, and σ_1, are also simple. Since the rotation operation can be implemented in reconfigurable hardware by only using routing resources, each of the aforementioned functions can be accommodated into a single LUT as shown in Fig. 7.11.

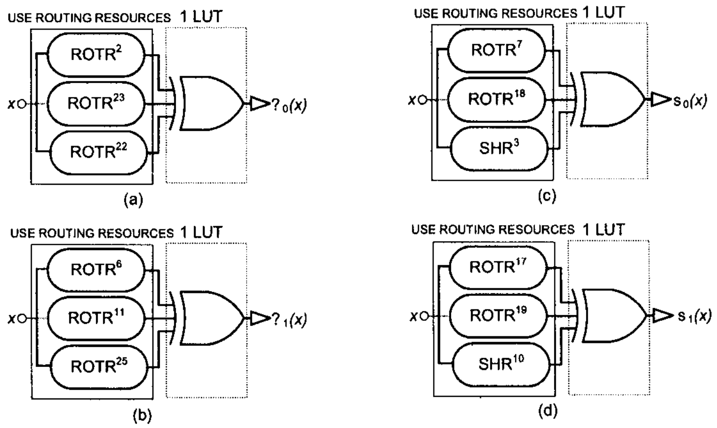

Fig. 7.11. Σ_0, Σ_1, σ_0, and σ_1 in Reconfigurable Hardware

7.4.4 Constants

Constants for SHA-1 and SHA-256 differ. On the other hand, SHA-384 and SHA-512, share the same constant values.

SHA-1

SHA-1 uses eighty 32-bit constant words K_0,K_1,, K_{79} which are given below, in hex format.

$$K_t = \begin{cases} 0x5a827999 & 0 \le t \le 19\\ 0x5a827999 & 20 \le t \le 39\\ 0x8f1bbcdc & 40 \le t \le 59\\ 0xca62c1d6 & 60 \le t \le 79 \end{cases}$$

SHA-256

SHA-256 uses sixty four 32-bit different constant words, $K_0, K_1, \ldots, K_{63}$. Those constants are extracted from the first 32 bits of the fractional parts of the first 64 prime numbers' cube roots. They are shown in hexadecimal format in Table 7.18.

Table 7.18. SHA-256 Constants

428a2f98	71374491	b5c0fbcf	e9b5dba5	3956c25b	59f111f1	923f82a4	ab1c5ed5
d807aa98	12835b01	243185be	550c7dc3	72be5d74	80deb1fe	9bdc06a7	c19bf174
e49b69c1	efbe4786	0fc19dc6	240ca1cc	2de92c6f	4a7484aa	5cb0a9dc	76f988da
983e5152	a831c66d	b00327c8	bf597fc7	c6e00bf3	d5a79147	06ca6351	14292967
27b70a85	2e1b2138	4d2c6dfc	53380d13	650a7354	766a0abb	81c2c92e	92722c85
a2bfe8a1	a81a664b	c24b8b70	c76c51a3	d192e819	d6990624	f40e3585	106aa070
19a4c116	1e376c08	2748774c	34b0bcb5	391c0cb3	4ed8aa4a	5b9cca4f	682e6ff3
748f82ee	78a5636f	84c87814	8cc70208	90befffa	a4506ceb	bef9a3f7	c67178f2

SHA-384 & SHA-512

SHA-384 and SHA-512 use eighty 64-bit different constant words $K_0, K_1, \ldots, K_{79}$. Those constants are extracted from the first 64 bits of the fractional parts of the first 80 prime numbers' cube roots. They are shown in hexadecimal format in Table 7.19.

7.4.5 Hash Computation

The main procedure for hash calculation in SHA-256, SHA-384, and SHA-512 is similar, only the word size varies. SHA-1 hash computation is however different. We can classify the hash calculation procedure of the SHA algorithm family into 3 major steps.

1. Define Word
2. Repeat Operation
3. Final Transformation

SHA-1

- Define Word: After performing message preprocessing for SHA-1, an i^{th} block message M_n^i $(0 \leq n \leq 15)$, is used to get 80 words for next steps as follows:

$$W_t = \begin{cases} M_t^i & 0 \leq t \leq 19 \\ ROTL^1(W_{t-3} \oplus W_{t-8} \oplus W_{t-16}) & 16 \leq t \leq 79 \end{cases}$$

Table 7.19. SHA-384 & SHA-512 Constants

428a2f98d728ae22	7137449123ef65cd	b5c0fbcfec4d3b2f	e9b5dba58189dbbc
3956c25bf348b538	59f111f1b605d019	923f82a4af194f9b	ab1c5ed5da6d8118
d807aa98a3030242	12835b0145706fbe	243185be4ee4b28c	550c7dc3d5ffb4e2
72be5d74f27b896f	80deb1fe3b1696b1	9bdc06a725c71235	c19bf174cf692694
e49b69c19ef14ad2	efbe4786384f25e3	0fc19dc68b8cd5b5	240ca1cc77ac9c65
2de92c6f592b0275	4a7484aa6ea6e483	5cb0a9dcbd41fbd4	76f988da831153b5
983e5152ee66dfab	a831c66d2db43210	b00327c898fb213f	bf597fc7beef0ee4
c6e00bf33da88fc2	d5a79147930aa725	06ca6351e003826f	142929670a0e6e70
27b70a8546d22ffc	2e1b21385c26c926	4d2c6dfc5ac42aed	53380d139d95b3df
650a73548baf63de	766a0abb3c77b2a8	81c2c92e47edaee6	92722c851482353b
a2bfe8a14cf10364	a81a664bbc423001	c24b8b70d0f89791	c76c51a30654be30
d192e819d6ef5218	d69906245565a910	f40e35855771202a	106aa07032bbd1b8
19a4c116b8d2d0c8	1e376c085141ab53	2748774cdf8eeb99	34b0bcb5e19b48a8
391c0cb3c5c95a63	4ed8aa4ae3418acb	5b9cca4f7763e373	682e6ff3d6b2b8a3
748f82ee5defb2fc	78a5636f43172f60	84c87814a1f0ab72	8cc702081a6439ec
90befffa23631e28	a4506cebde82bde9	bef9a3f7b2c67915	c67178f2e372532b
ca273eceea26619c	d186b8c721c0c207	eada7dd6cde0eb1e	f57d4f7fee6ed178
06f067aa72176fba	0a637dc5a2c898a6	113f9804bef90dae	1b710b35131c471b
28db77f523047d84	32caab7b40c72493	3c9ebe0a15c9bebc	431d67c49c100d4c
4cc5d4becb3e42b6	597f299cfc657e2a	5fcb6fab3ad6faec	6c44198c4a475817

- Repeat Operation: A single operation for SHA-1 is shown in Fig. 7.12 which must be repeated 80 times. Let us recall that for the first sub block message, initial values for words a, b,c,d, and e are provided by the algorithm. For the next message sub-blocks, the output hash value of an i^{th} message block serves as initial vector for the hash computation process of the next sub block message. The symbol K_t represents SHA-1 constant values.

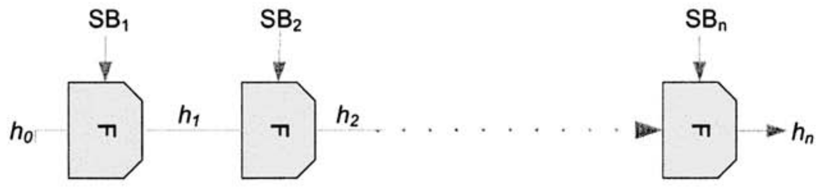

Fig. 7.12. Single Operation for SHA-1

- Final Transformation: Final transformation is simply the addition (modulo 2^{32}) of the initial hash value with the final output hash value of the N^{th} sub block message. A 160-bit hash of the message is then obtained by concatenating five 32-bit words, namely,

$$a \,||\, b \,||\, c \,||\, d \,||\, e$$

SHA-256

- Define Word: After performing message preprocessing for SHA-256, an i^{th} block message M_n^i $(0 \leq n \leq 15)$, is used to get 64 words for next steps as follows[2]:

$$W_t = \begin{cases} M_t^i & 0 \leq t \leq 19 \\ \sigma_1(W_{t-2}) + W_{t-7} + \sigma_0(W_{t-15}) & 16 \leq t \leq 63 \end{cases}$$

- Repeat Operation: A single operation for SHA-256 is shown in Fig. 7.13 which is repeated for 60 times. Similarly as in SHA-1, for the first sub block message, initial values for 8 words a, b,c,d,e,f,g, and h are provided by the algorithm. For next message blocks, output hash values for an i^{th} block message serve as initial vectors for hash calculating process on next sub block message. The symbol K_t represents constant values for SHA-256.

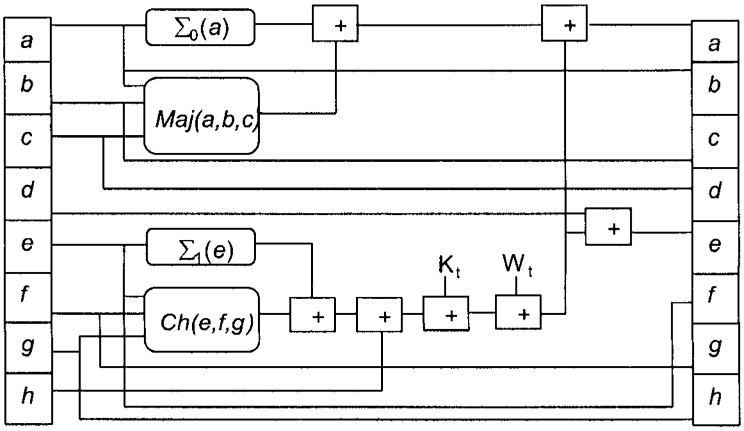

Fig. 7.13. Single Operation for SHA-256

- Final Transformation: Final transformation is simply the addition (modulo 2^{32}) of the initial hash values with the final output hash values of N^{th} message sub block. A 256-bit hash of the message is then obtained by concatenating eight 32-bit words, namely,

$$a \,||\, b \,||\, c \,||\, d \,||\, e \,||\, f \,||\, g \,||\, h$$

[2] The operations $\oplus$ and $+$, must not be mixed.

SHA-384

- Define Word: After performing message preprocessing for SHA-384, an i^{th} block message M_n^i $(0 \leq n \leq 15)$, is used to get 80 words for the next steps as follows[3],

$$W_t = \begin{cases} M_t^i & 0 \leq t \leq 19 \\ \sigma_1(W_{t-2}) + W_{t-7} + \sigma_0(W_{t-15}) & 16 \leq t \leq 63 \end{cases}$$

  Here addition is performed modulo 2^{64}.
- Repeat Operation: A single operation for SHA-384 is similar to that of SHA-256 as shown in Fig. 7.13. The difference lies in the number of repetitions which are 80, instead of the 60 repetitions of SHA-256.
- Final Transformation: Final transformation consists on the addition (modulo 2^{64}) of the initial hash values with the final output hash values of N^{th} sub block message. A 384-bit message digest is then obtained by truncating the last 2 words. The first six 64-bit words are concatenated as follows.

$$a \,||\, b \,||\, c \,||\, d \,||\, e \,||\, f$$

SHA-512

The process of hash computation for SHA-512 is quite similar to that of SHA-384. There are only two exceptions. The first one is due to loading the initial values for the 8 words a, b,c,d,e,f,g, and h, which are different for both SHA-384 and SHA-512. The second difference is that a 512-bit message digest is obtained by concatenating all 8 words. Last 2 words are not truncated as it is in the case of SHA-384.

$$a \,||\, b \,||\, c \,||\, d \,||\, e \,||\, f \,||\, g \,||\, h$$

7.5 Hardware Architectures

The main moral of the preceding Sections is that hash function computation is iterative in nature. To calculate hash values, several rounds must be performed where each round comprises a certain number of steps. The output of a step serves as input to the next step and the output of a round serves as the input of the next round.

That characteristic does not prevent us from designing a fully pipeline or sub pipeline architecture for hash functions. Let us recall that the input message M is divided into N blocks. Hash computation of a new block cannot start until the hash computation of the previous block has been fully completed. The hash values (output) of the first block are now the initial values

[3] It is noticed that the word size for SHA-384 is 64-bit as compared to SHA-256 which is 32-bit long.

for the hash computation of the second block message. That restricts us from start processing the second block although only a single stage is active and all others are idle during hash computation.

However, different strategies have been proposed by designers in order to improve the data flow at different stages of the design so that high speed gains can be obtained. The different design strategies are discussed in the rest of this Section.

7.5.1 Iterative Design

An iterative design is a natural approach for the implementation of hash functions on hardware platforms. Fig. 7.14 presents an iterative approach for implementing hash algorithms in hardware.

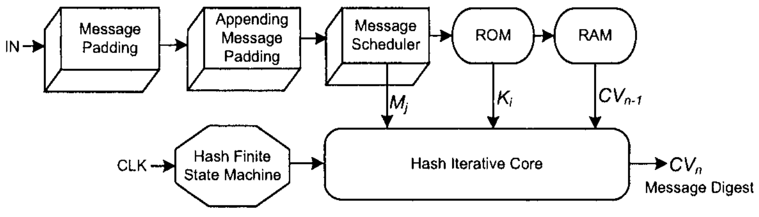

Fig. 7.14. Iterative Approach for Hash Function Implementation

The input message is formatted according to the algorithm requirements in two steps. Those are message padding, and then appending the message length on it. Message scheduler shall provide a sub block or a word derived from some sub blocks for any given algorithm step. Constants provided by the algorithm can be stored in a memory block (ROM). The initial hash values are required till the end of one iteration of the algorithm. This is in order to perform the final transformation (simple XOR with the final output of the iteration). Hence, at the end of a given iteration, partial results must update the input parameters for the next iteration. BRAMs can be used for accomplishing this operation.

The block labeled: "Hash Iterative Core" in Fig. 7.14, includes all logical steps needed for accomplishing a particular compression function computation. The exact sequence of those logical steps (i.e., when should they be executed and with which parameters), is synchronized by the module labeled "Hash Finite State Machine" block. Clearly, the main building blocks of Fig. 7.14 can be altered/combined/modified using different techniques according to the characteristics of the target device and the hash algorithm in hand.

7.5.2 Pipelined Design

In pipeline architectures, registers are provided at different stages of the algorithm. At each clock cycle, the output of a stage is shifted to the next stage. Thus, at the first clock cycle, one input block should be loaded. At the next clock cycle, a second block must be loaded and so on. Once the pipeline is filled, i.e., the final stage outputs a data, then an output value will be ready at each clock cycle.

Pipeline is a fast approach but cost has to be paid in terms of hardware resources. Unfortunately, that approach cannot be fully utilized for hash function computation due to the inherent dependencies. As it was explained, the second iteration cannot be started until the computations for first iteration have been completed. However a sort of pipelining can be achieved for different operations of the similar stage.

7.5.3 Unrolled Design

Unrolled design approach is a useful technique used on the implementation of hash algorithms in order to improve their performance on time. In this approach, all or part of the stages of a hash algorithm are unrolled as is shown in Fig. 7.15a. That however produces long critical paths which causes undesirable long path delays in the circuit. Most designers therefore prefer to unroll some k stages and then to cascade them for the implementation of the whole algorithm as is shown in Fig. 7.15b.

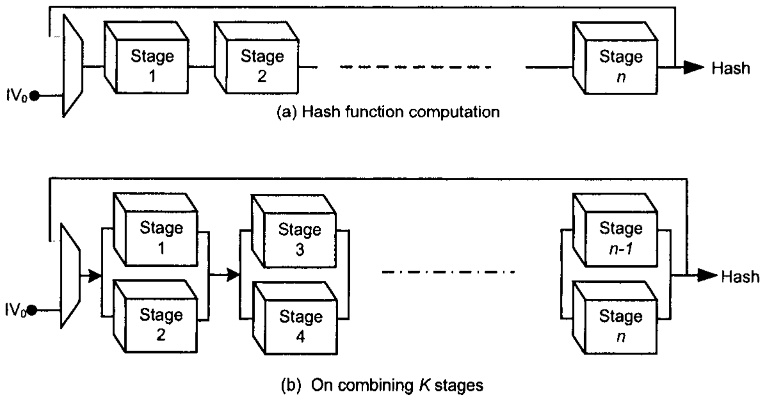

Fig. 7.15. Hash Function Implementation (a) Unrolled Design (b) Combining k Stages

7.5.4 A Mixed Approach

Designing circuits with long critical paths is not useful especially if the target devices are FPGAs. The propagation of long time delays usually implies a performance diminishing. However some registers can be provided as interface buffers between neighbor stages of the hash algorithm. That can be also helpful for producing a more compact design, which will help the mapping synthesis tool. Another enhancement can be made by combining an unrolled design structure with the provision of registers between different stages as shown in Fig. 7.16.

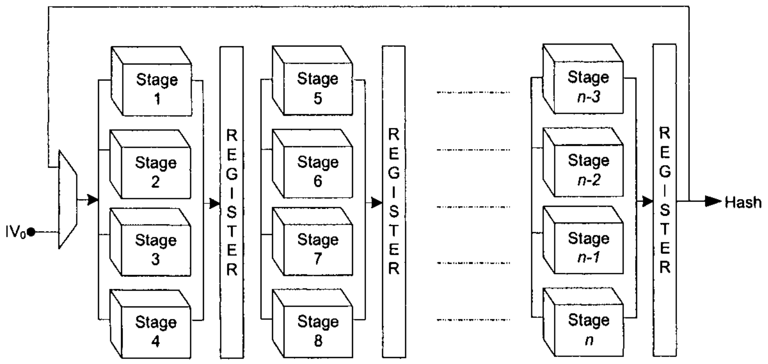

Fig. 7.16. A Mixed Approach for Hash Function Implementation

7.6 Recent Hardware Implementations of Hash Functions

Various hardware implementations of hash algorithms have been reported in literature. Some of them focus on speed optimization while others concentrate on saving hardware resources. Some authors have also tried to exploit parallelism in operations whenever this can be done. Some designs present a tradeoff between time and hardware resources. It has been shown that by adding few registers or few memory units, considerable timing improvements can be obtained.

In the rest of this Section we review some of the most representative hash function hardware designs recently reported. In total, we review six hash function algorithms, namely, MD4, MD5, SHA-1, RIPEMD-160, SHA-2 and Whirpool.

MD4

A single MD4 FPGA architecture has been reported in the open literature [328]. The distinct feature of this design is to try to exploit as much parallelism and pipelining for the MD4 hash algorithm as possible. That design implements arithmetic, logic and circular shift operation using a pipelined parallel processor. It takes 94.07 μS to compute the message digest of a 512-bit input message block at 6.67 MHz frequency consuming only 252 CLB slices.

Table 7.20. MD5 Hardware Implementations

Author(s)	Target Device	Cost	Freq. MHz	Cycles	T† Mbps	T/S
Fastest ASIC MD5 Cores						
Satoh et al. [312]	0.13μm ASIC	17.7K gates	277.8	68	2091	0.117
Compact ASIC MD5 Cores						
Satoh et al. [312]	0.13μm ASIC	10.3K gates	133.3	68	1004	0.097
Helicon [358]	0.18μm ASIC	16K gates	145	65	1140	0.072
Sandra [71]	0.6μm ASIC	10.9K gates + RAM	59	206	146	0.013
Fastest FPGA MD5 Cores						
Jarvinen et al. [156]	Virtex-II XC2V4000-6	11.5K(10) slices(RAM)	75.5	66	5857	0.509
Compact FPGA MD5 Cores						
Helicon [358]	Virtex-II	613(1)	96	66	744	1.213
		slices(RAM)				
Other FPGA MD5 Cores						
Jarvinen et al. [156]	Virtex-II XC2V4000-6	5.7K(0) 647(2) slices(RAM)	80.7 75.5	66 66	2395 586	0.417 0.905
Helicon [358]	Spartan3	630(1) slices(RAM)	63	66	488	0.774
Sandra [71]	Virtex XCV300E	2008 slices	42.9	206	107	0.053
Kang et al. [166]	Apex EP20K1000E	10.5K logic cells	18	65	142	0.0134
Deepak. et al. [65]	Virtex XCV1000-6	880(2) slices(RAM)	21	65	165	0.187

† Throughput

MD5

A considerable number of MD5 hardware implementations have been reported in the open literature. Table 7.20 presents some selected designs. However, due to the availability of a large number of FPGA devices by different manufacturers, with different logic complexity within the basic building block, a comparison of different hash cores becomes complicated.

The ASIC MD5 design in [312] is the fastest one in its category, with a throughput of 2.09 Gbps at a cost of 17,764 gates on a $0.13\mu m$ chip.

The authors in [156] designed several MD5 architectures by unrolling a variable number of MD5 stages. A fully unrolled MD5 architecture is their fastest design, achieving a throughput of 5.8 Gbps by occupying 11498 slices plus 10 BRAMs on a Xilinx Virtex-II XC2V4000-6.

A commercially available MD5 core designed by [358] is a compact design that occupies only 630 slices plus 1 BRAM and reports a throughput of 744 Mbps on a Xilinx Virtex-II device. The throughput over area factor (our figure of merit for measuring efficiency) achieved in [358] is the best one of all designs considered in Table 7.20.

Other MD5 architectures on different FPGA chips using different design approaches are also reported in Table 7.20.

SHA-1

Numerous SHA-1 FPGA implementations have been reported in the literature. A representative group of them are shown in Table 7.21.

The authors in [312] presented two SHA-1 architectures in ASIC hardware, one of them is the fastest architecture reported in the literature, achieving a throughput of 2 Gbps by utilizing 9859 gates in a $0.13\mu m$ chip.

In the reconfigurable hardware category, the fastest design, reported in [67] achieves a throughput of 899.8 Mbps. That is also a compact design with the best throughput over area performance.

A SHA-1 architecture in [120] is the 2^{nd} fastest FPGA core. It utilizes carry save adders to speed up multi-operand additions and to minimize delays with carry propagation. This design reduces the number of operands in a round by pre-computing addition of Constants (K) and Words(W) $(K_t + W_t)$ and also it eliminates the final round which is incorporated as a conditional addition within a round. The throughput for this design is reported as 462 Mbps when operating at a 75.8 MHz clock frequency.

The most compact design for SHA-1 was presented in [71] using as a target device a Xilinx V300E. It proposes a pipelined parallel structure by implementing two arithmetic logic units for SHA-1, achieving a throughput of 119 Mbps at a 59 MHz clock frequency.

The design in [404] utilizes 1622 slices on an Altera EPIK100QC208-1 achieving a throughput of 268.99 Mbps. That is another compact hardware SHA-1 core on Altera devices.

Table 7.21. Representative SHA-1 hardware Implementations

Author(s)	Target Device	Hardware	Freq. MHz	Cycles	$T^{\dagger}$ Mbps	T/S
Fastest ASIC SHA-1 Cores						
Satoh et al [312]	0.13μm ASIC	9.9K gates	333.3	85	2006	0.203
Compact ASIC SHA-1 Cores						
Satoh et al [312]	0.13μm ASIC	7.9K gates	154.3	85	929	0.116
Helicon [358]	0.18μm ASIC	20K gates	166	81	1000	0.050
Sandra [71]	0.6μm ASIC	10.9K + RAM gates	59	255	119	0.011
Compact & Fastest FPGA SHA-1 Cores						
Diez et al [67]	Virtex-II XC2V3000	1.55K slices	38.6	22	899.8	0.580
Grembowski et al [120]	Virtex XCV1000-6	2.2K slices	75.76	84	462	0.210
Other FPGA SHA-1 Cores						
Sandra [71]	Virtex V300E	2.0K slices	42.9	255	86	0.042
Zibin et al [404]	Apex EPIK100Q	1.6K logic cells	43.08	82	268.99	0.165
Kang et al [166]	Apex EP20K1000	10.5K logic cells	18	81	114	0.011
Sklavos [332]	Virtex XCV300	2.6K slices	37		233	0.089

† Throughput

Additionally, there exist other SHA-1 cores [67, 404, 166, 332] which propose some effective techniques to save hardware resources and to increase time factor. In [166], a significant saving of resources was achieved. This design implements a switching matrix by using multiplexers for an appropriate word (W) selection. It can operate at 18 MHz and achieves a throughput of 114 Mbps.

The SHA-1 implementation in [332] was used as a pseudo-random number generator. It is actually a VLSI architecture which was first captured in VHDL and synthesized on FPGAs. That design allows a system frequency of 37 MHz and can run at the rate of 233 Mbps.

Finally, the SHA-1 core in [404] explores three Altera FPGA grades for the same SHA-1 code.

RIPEMD-160

Table 7.22 presents two FPGA architectures for RIPEMD-160, which were implemented on devices made by different manufacturers. The design in [249] is a unified architecture in Altera EPF10K50SBC356-1 for two different hash algorithms:RIPEMD-160 and MD5. That design achieves a throughput over 200 Mbps for MD5 and 84 Mbps for RIPEMD-160 when operating at 26.66 MHz and it stands as the compact and the fastest RIPMD architecture in FPGAs. In [71], a RIPEMD-160 FPGA implementation on Xilinx V300E can run at a 42.9 MHz frequency and achieves a data rate of 89 Mbps.

In ASIC hardware, the fastest RIPEMD architecture is due to [312]. That design can run at 1.442 Gbps by occupying 24755 gates on a $0.13\mu m$ chip.

Table 7.22. Representative RIPEMD-160 FPGA Implementations

Author(s)	Target Device	Hardware	Freq. MHz	Cycles	T[†] Mbps	T/S
Fastest ASIC RIPEMD Cores						
Satoh et al [312]	$0.13\mu m$ ASIC	24775 gates 17446 gates	270.3 142.9	96 96	1442 762	0.058 0.044
Sandra [71]	$0.6\mu m$ ASIC	10,900 gates + RAM	59	337	89	0.008
Compact & Fastest FPGA RIPEMD Cores						
Ng et al [249]	Apex EPF10K50S-1	1964 logic elements	26.66	162	84	0.042
Sandra [71]	Virtex V300E	2008 slices	42.9	337	65	0.032

† Throughput

SHA-2

Table 7.23 shows several representative SHA-2 hardware cores reported in the open literature.

Authors in [312] reported four ASIC architectures for SHA-224, SHA-256, SHA-384, and SHA-512 implemented on a $0.13\mu m$ chip. The fastest among them is the SHA-512 architecture that achieves a throughput of 2.9 Gbps by using 27297 gates. That is also the fastest ASIC hardware architecture of any SHA-2 family of hash algorithms.

The fastest FPGA SHA-2 architectures have been proposed in [222]. It achieves a throughput of 1466 Mbps on a Xilinx Virtex-II device. The architecture employed for that SHA-2 (512-bit) design consisted on a two-step (2x) unrolled implementation. Authors in [222] essayed six variants of the same design which are named as SHA2 (256) basic, SHA2 (256) 2x-unrolled, SHA2 (256) 4x-unrolled, SHA2 (512) basic, SHA2 (512) 2x-unrolled and SHA2 (512)

Table 7.23. Representative SHA-2 FPGA Implementations

Author(s)	Target Device	Hardware	Freq. MHz	Cycles	T† Mbps	T/S
ASIC SHA-2 Cores						
Satoh et al [312]						
SHA-224	0.13μm ASIC	11484 gates	154.1	72	1096	0.095
SHA-256	0.13μm ASIC	15329 gates	333.3	72	2370	0.154
SHA-384	0.13μm ASIC	23146 gates	125.0	88	1455	0.062
SHA-512	0.13μm ASIC	27297 gates	250.0	88	2909	0.106
Helicon [358] SHA-256	0.18μm ASIC	22K gates	200	65	1575	0.072
Fastest FPGA SHA-2 Cores						
McEvoy [222] SHA-2(512)	Virtex-II XC2V2000	4107 slices	65.893	46	1466	0.357
Compact FPGA SHA-2 Cores						
Sklavos et al [333] SHA-2(256)	Virtex XCV200-6	1060 slices	83		326	0.307
Other FPGA SHA-2 Cores						
Sklavos et al [333] SHA-2(384)	Virtex XCV200-6	1966 slices	74		350	0.178
Sklavos et al [333] SHA-2(512)	Virtex XCV200-6	2237 slices	75		480	0.214
McLoone et al [224] SHA-2(384)	Virtex XCV600E-8	2914 slices + 2 BRAMs	38	80	479	0.164
McLoone et al [224] SHA-2(512)	Virtex XCV600E-8	2914 slices 2 BRAMs	38	80	479	0.164
McEvoy [222] SHA-2(256)						
(Basic)	Virtex-II XC2V2000	1373 slices	133.06	68	1009	0.734
(2x-unrolled)	Virtex-II XC2V2000	2032 slices	73.975	38	996.7	0.490
(4x-unrolled)	Virtex-II XC2V2000	2898 slices	40.833	23	908.9	0.313
McEvoy [222] SHA-2(512)						
(Basic)	Virtex-II XC2V2000	2726 slices	109.03	84	1329	0.487
(4x-unrolled)	Virtex-II XC2V2000	5807 slices	35.971	27	1364	0.234

† Throughput

4x-unrolled. Those architectures optimize time performances by combining pipelining and unrolling techniques.

In [333], a common architecture is customized for three SHA2 algorithms: SHA2 (256), SHA2 (384) and SHA2 (512). The design compares three implementations in terms of operating frequency, throughput and area-delay product. Among them, SHA2 (256) FPGA implementation consumes least hardware resources in the literature, achieving a throughput of 326 Mbps on a Xilinx V200PQ240-6.

In [224], a single chip FPGA implementation is also presented for SHA2 (384) and SHA2 (512). That architecture optimizes time factor and hardware area by using shift registers for message scheduler and compression block. Similarly, block select RAMs (BRAMs) are used to store the compression function constants.

Table 7.24. Representative Whirlpool FPGA Implementations

Author(s)	Target Device	Hardware	Freq. MHz	Cycles	T† Mbps	T/S
Fastest FPGA Whirlpool Cores						
McLoone et al [226] 2×unrolled	Virtex-4 X4VLX100	13210 slices	47.8		4896	0.370
Kitsos et al [173] LUT based Time optimized	Virtex XCV1000E	5585 slices	87.5	10	4480	0.802
Compact FPGA Whirlpool Cores						
Pramstaller et al [274]	Virtex-2P XC2VP40	1456 slices	131		382	0.262
Other FPGA Whirlpool Cores						
Kitsos et al [173] Boolean expression based	VirtexE XCV1000E	3815 slices	75	20	1920	0.503
Kitsos et al [173] LUT based	VirtexE XCV1000E	3751 slices	93	20	2380	0.634
Kitsos et al [173] Boolean expression based Time optimized	VirtexE XCV1000E	5713 slices	72	10	3686	0.645
McLoone [226]	Virtex-4 X4VLX100	4956 slices	93.56		4790	0.966

† Throughput

Whirlpool

Table 7.24 lists various Whirlpool FPGA-based architectures. The fastest Whirlpool core has been reported in [226]. That is a 2 stages (2x) unrolled Whirlpool architecture implemented on a Xilinx Virtex-4 which achieves a throughput of 4896 Mbps by consuming 13210 CLB slices.

Another Whirlpool core showing similar throughput to the design in [226] is due to [173] which reports a throughput of 4480 Mbps on a Xilinx XCV1000 by occupying 5585 CLB slices and also some dedicated memory modules. Three more variants of that design are also presented. Those architectures implement Whirlpool mini boxes by using Boolean expressions, referred to as BB (Boolean expressions Based) and by using FPGA LUTs, referred to as LB (LUT Based) respectively. Let us call them as Whirlpool BB and Whirlpool LB. Both Whirlpool BB and Whirlpool LB can operate at rates of 1920 Mbps and 2380 Mbps. Both architectures are further optimized for time, increasing throughputs to 3686 Mbps and 4480 Mbps.

In contrast to the aforementioned architectures, a compact FPGA implementation of Whirlpool hash function was reported in [274]. That architecture focuses on saving considerable hardware resources by using LUT-based RAM for Whirlpool state. Authors report a hardware cost of just 1456 CLB slices achieving a data rate of 382 Mbps.

7.7 Conclusions

In this chapter, various popular hash algorithms were described. The main emphasis on that description was made on evaluating hardware implementation aspects of hash algorithms.

MD5 description included in this Chapter can be regarded as a step by step example of how intermediate values are being updated during algorithm execution. We have mentioned that MD5 design methodology has a strong influence in almost all modern hash functions. The explanation provided for SHA family of hash algorithms can be regarded as an evidence that the structure of current hash algorithms borrows basic rules and principles from their predecessors.

A fair number of hash function implementations in reconfigurable Hardware have been reported so far. Those architectures do not pretend to be a universal solution for all the universe of hash applications such as, secure web traffic (https /SSL), encrypted e-mail(PGP, S/MIME), digital certificates, cryptographic document authenticity, secure remote access (ssh/sftp), etc.

However, the usage of reconfigurable hardware for hash function implantations can provide a unique benefit of reconfiguring customized hardware architecture according to the specifications of end users. Furthermore, given the fact that most hash functions are enduring difficult times, where several emblematic hash functions have been critically attacked, new security patches could be easily incorporated.