



Chapter 11

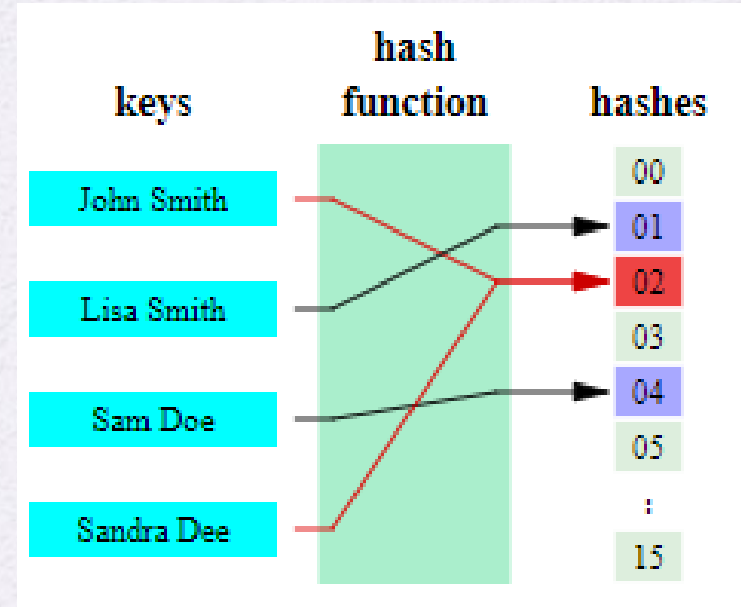
Cryptographic Hash Functions

Outline

- Applications of cryptographic hash functions
- Requirements and security
- Hash functions based on cipher block chaining
- Secure hash algorithm (SHA)

Hash Function Definition

- Generates **fixed-length** hash value which is **computed from the plaintext**
- Not reversible: it is impossible for either the **contents** or **length** of the plaintext to be recovered.
- hash value is less (in number of bits) than plaintext.
- One-way property:
 - hard to determine plain text.
 - more than one plain text can map to one hash



$$h = H(M)$$

hash value

hash function

hashed message

Hash Function: formal definition

- A hash function **H** is an algorithm which:
 - accepts a variable-length block of data **M** as input and produces a fixed-size hash value **h**, where : $h = H(M)$
 - Principal usage for hash is data integrity
- A “good” hash function has the property that
 - the results of applying the function to a large set of inputs will produce outputs that are **evenly distributed** and apparently **random**.
 - A change to any bit or bits in **M** results, with high probability, in a change to the hash code.

Hash Function: formal definition

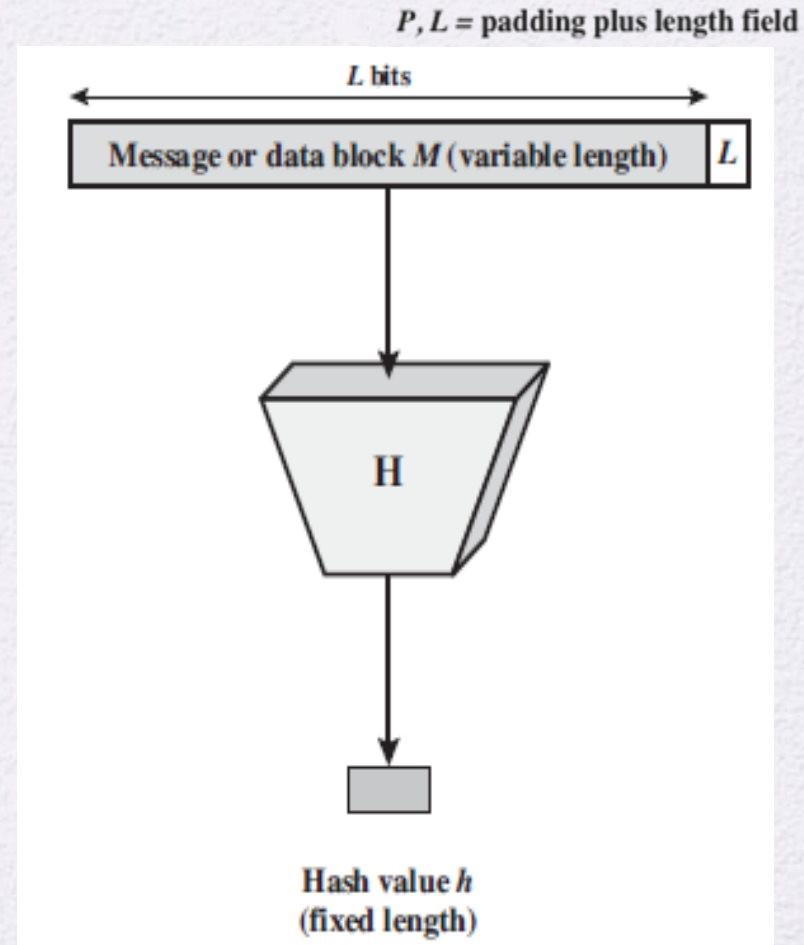
- Cryptographic hash function
 - An algorithm for which it is **computationally infeasible** to find either:
 - (a) a data object that maps to a pre-specified hash result (the one-way property)
 - (b) two data objects that map to the same hash result (the collision-free or collision-resistant property)

Collision Resistant

- Collision resistance is a property of cryptographic hash functions
- A hash function H is collision resistant
 - if it is hard to find two inputs that hash to the same output; that is, two inputs a and b such that $H(a) = H(b)$, and $a \neq b$

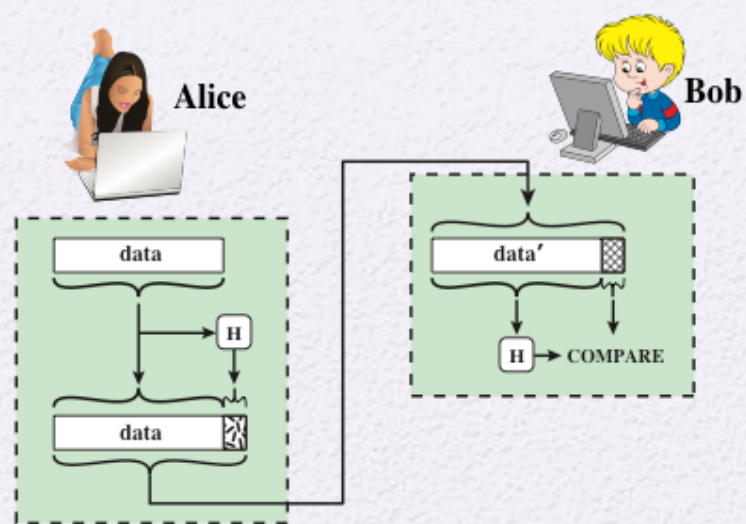
Block Diagram of Hash Function

- $h = H(M)$
- Typically, the input is padded out to an integer multiple of some fixed length (e.g., 1024 bits), and the padding includes the value of the length of the original message in bits.

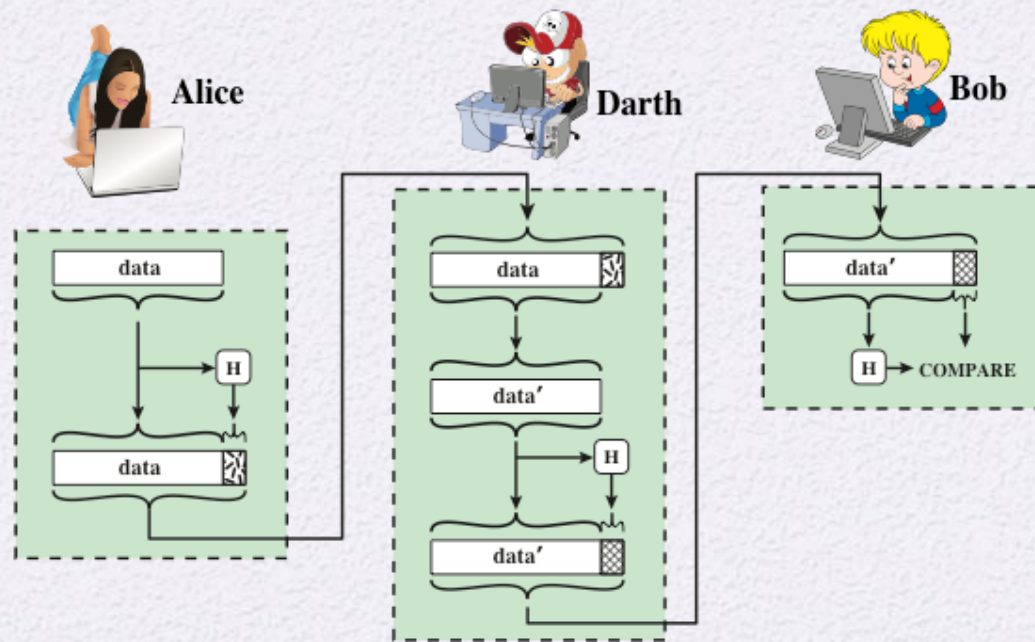


Applications of Hash Function

- **Message Authentication:** authenticates/confirms that the message has not been modified.
 - verify the **integrity** of a message.
 - message **authentication** is achieved using a message authentication code (MAC), also known as a keyed hash function.
 - $MAC = \text{Hash} + \text{symmetric Key}$
- **Digital Signatures:** verifies the authenticity of a message i.e. has not been tampered with.
 - Digital Signature = **Hash** + **asymmetric key**
 - Anyone who knows the user's public key can verify the integrity of the message that is associated with the digital signature.
- **one-way password file**
- **intrusion detection and virus detection**
- **Message Authentication vs Digital signature**
 - Message Authentication: receiver does not distinguish between senders sharing same key. Digital signature does.
 - Message authentication is computed faster than digital signature.



(a) Use of hash function to check data integrity

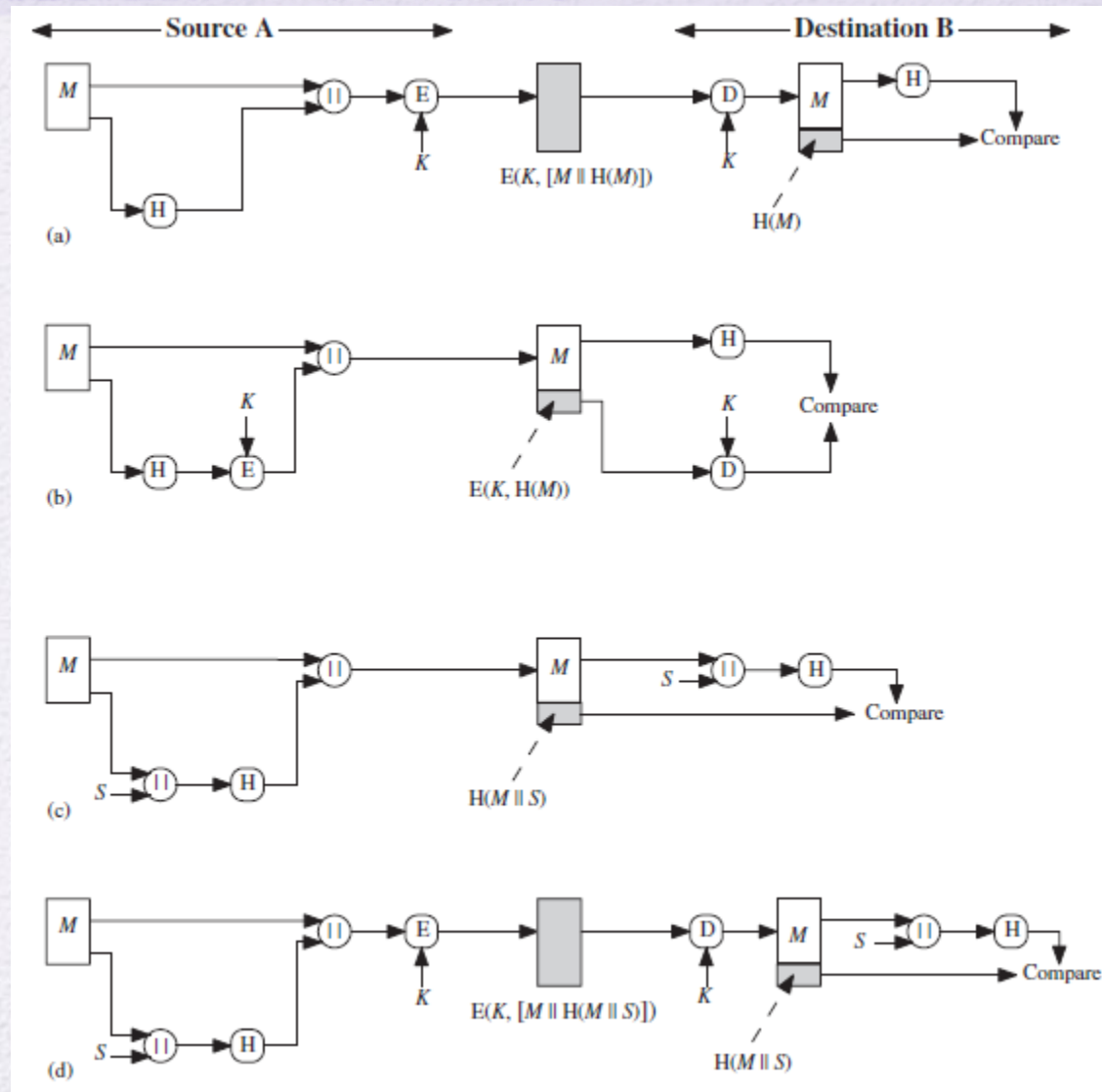


(b) Man-in-the-middle attack

Figure 11.2 Attack Against Hash Function

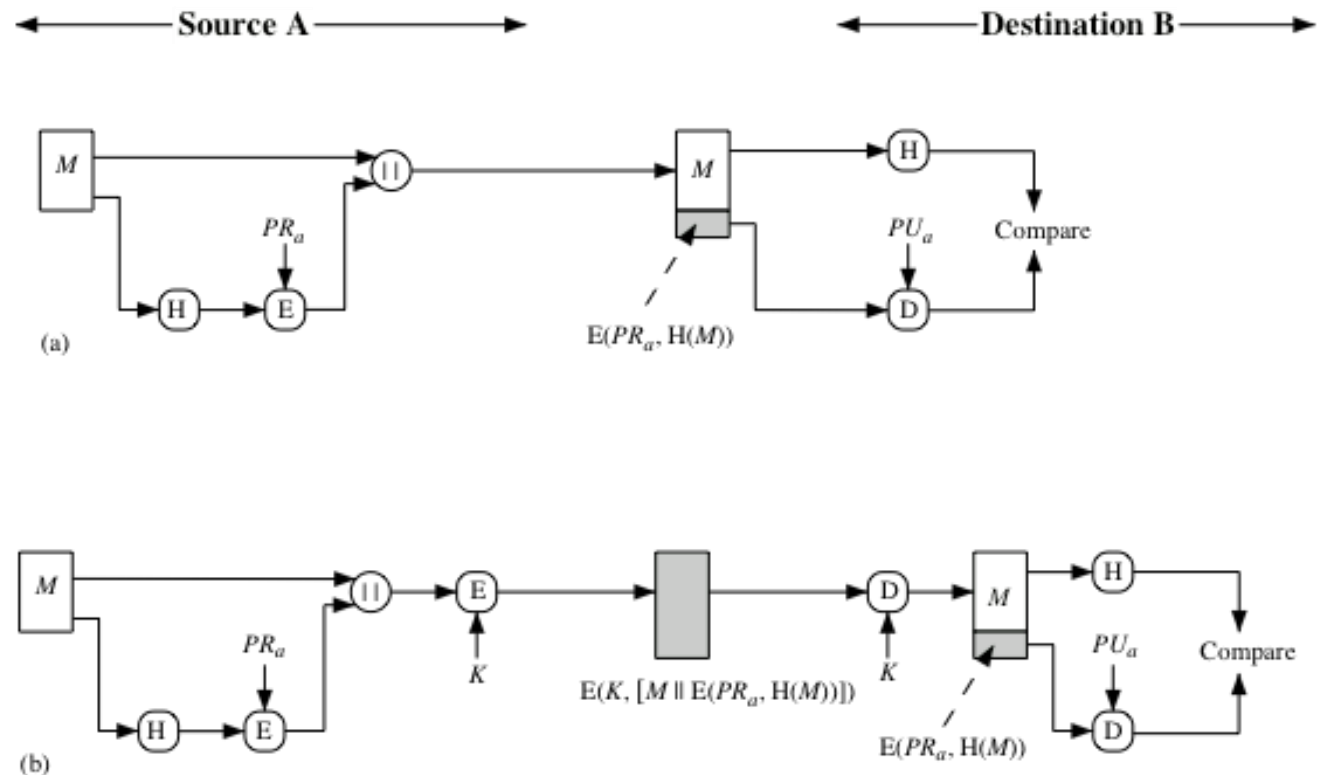
Examples of the Use of Hash Code for Message Authentication

- The message plus concatenated hash code is encrypted using symmetric encryption. The hash code provides **authentication**. Because encryption is applied to the entire message plus hash code, this **provides message authentication + confidentiality**.
- Only the hash code is encrypted, using symmetric encryption. This reduces the processing burden. So this **provides message authentication with no confidentiality**.
- Message authentication with no encryption using shared secret S**. A computes the hash value over the concatenation of M and S and appends the resulting hash value to M. Because the secret value itself is not sent, an opponent cannot modify an intercepted message and cannot generate a false message. This **provides message authentication + resists attacks, with no confidentiality**.
- Confidentiality** can be added to the approach of method (c) by encrypting the entire message plus the hash code. This **provides message authentication + resists attacks + confidentiality**.



Examples of Digital Signatures

- a) The **hash code is encrypted**, using public-key encryption with the sender's private key. This provides **authentication**. It also provides a digital signature, because only the sender could have produced the encrypted hash code. This is the digital signature technique.
- b) If **confidentiality** as well as a **digital signature** is desired, then the message plus the private-key encrypted hash code can be encrypted using a symmetric secret key. This is a common technique.



Digital Signature

**Digital Signature
+ confidentiality**

Other Hash Function Uses

Commonly used to create a one-way password file

When a user enters a password, the hash of that password is compared to the stored hash value for verification

This approach to password protection is used by most operating systems

Can be used for intrusion and virus detection

Store $H(F)$ for each file on a system and secure the hash values

One can later determine if a file has been modified by recomputing $H(F)$

An intruder would need to change F without changing $H(F)$

Can be used to construct a pseudorandom function (PRF) or a pseudorandom number generator (PRNG)

A common application for a hash-based PRF is for the generation of symmetric keys

Two Simple Hash Functions

- Consider two simple insecure hash functions that operate using the following general principles:
 - The input is viewed as a sequence of n -bit blocks
 - The input is processed one block at a time in an iterative fashion to produce an n -bit hash function
- Bit-by-bit exclusive-OR (XOR) of every block
 - $C_i = b_{i1} \text{ xor } b_{i2} \text{ xor } \dots \text{ xor } b_{im}$
 - Produces a simple parity for each bit position and is known as a longitudinal redundancy check
 - Reasonably effective for random data as a data integrity check
- Perform a one-bit circular shift on the hash value after each block is processed
 - Has the effect of randomizing the input more completely and overcoming any regularities that appear in the input

Two Simple Hash Functions

Function 1: XOR

One of the simplest hash functions is the bit-by-bit exclusive-OR (XOR) of every block. This can be expressed as

$$C_i = b_{i1} \oplus b_{i2} \oplus \dots \oplus b_{im}$$

where

C_i = i th bit of the hash code, $1 \leq i \leq n$

m = number of n -bit blocks in the input

b_{ij} = i th bit in j th block

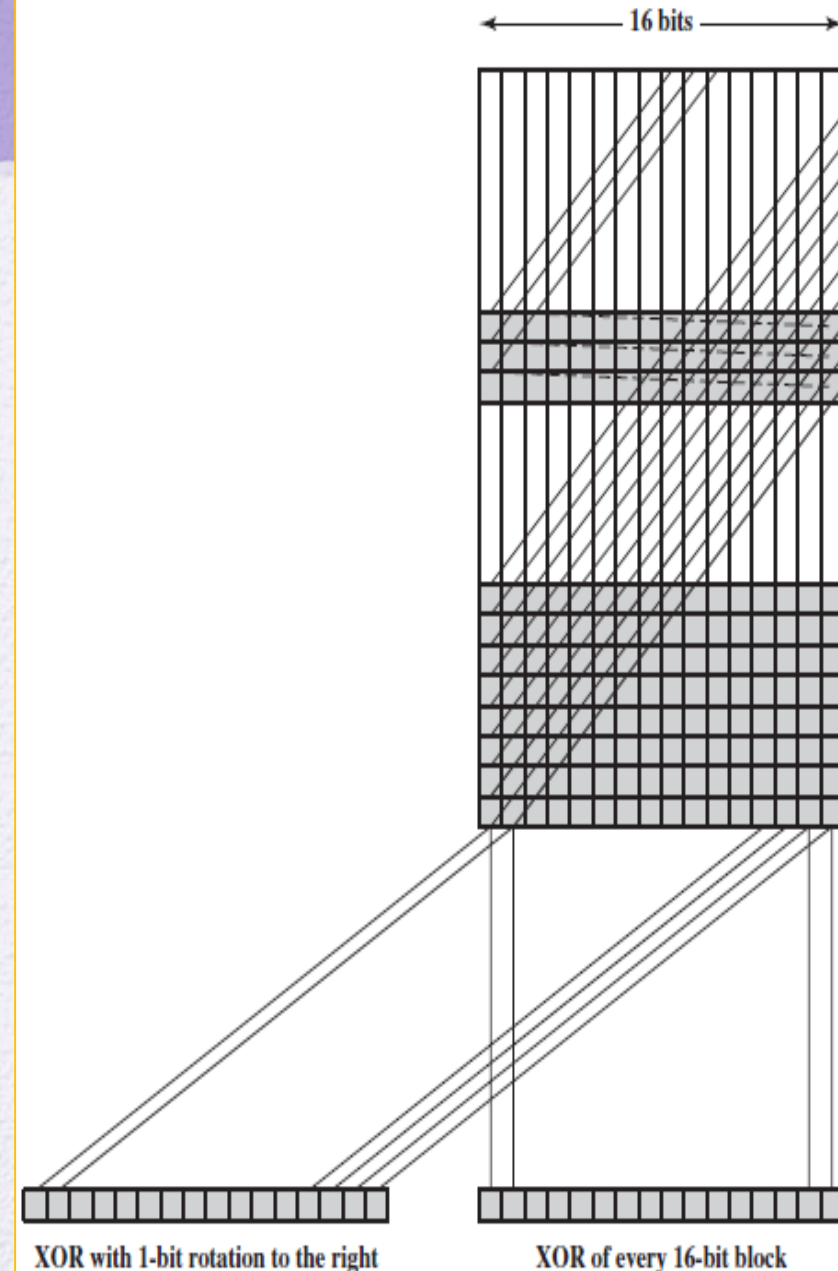
\oplus = XOR operation

Function 2: Rotated XOR


A simple way to improve matters is to perform a one-bit circular shift, or rotation, on the hash value after each block is processed. The procedure can be summarized as follows.

1. Initially set the n -bit hash value to zero.
2. Process each successive n -bit block of data as follows:
 - a. Rotate the current hash value to the left by one bit.
 - b. XOR the block into the hash value.

Given a message, it is an easy matter to produce a new message that yields that hash code: Simply prepare the desired alternate message and then append an n -bit block that forces the new message plus block to yield the desired hash code.



Requirements for a Cryptographic Hash Function H



Requirement	Description
Variable input size	H can be applied to a block of data of any size.
Fixed output size	H produces a fixed-length output.
Efficiency	$H(x)$ is relatively easy to compute for any given x , making both hardware and software implementations practical.
Preimage resistant (one-way property)	For any given hash value h , it is computationally infeasible to find y such that $H(y) = h$.
Second preimage resistant (weak collision resistant)	For any given block x , it is computationally infeasible to find $y \neq x$ with $H(y) = H(x)$.
Collision resistant (strong collision resistant)	It is computationally infeasible to find any pair (x, y) such that $H(x) = H(y)$.
Pseudorandomness	Output of H meets standard tests for pseudorandomness.

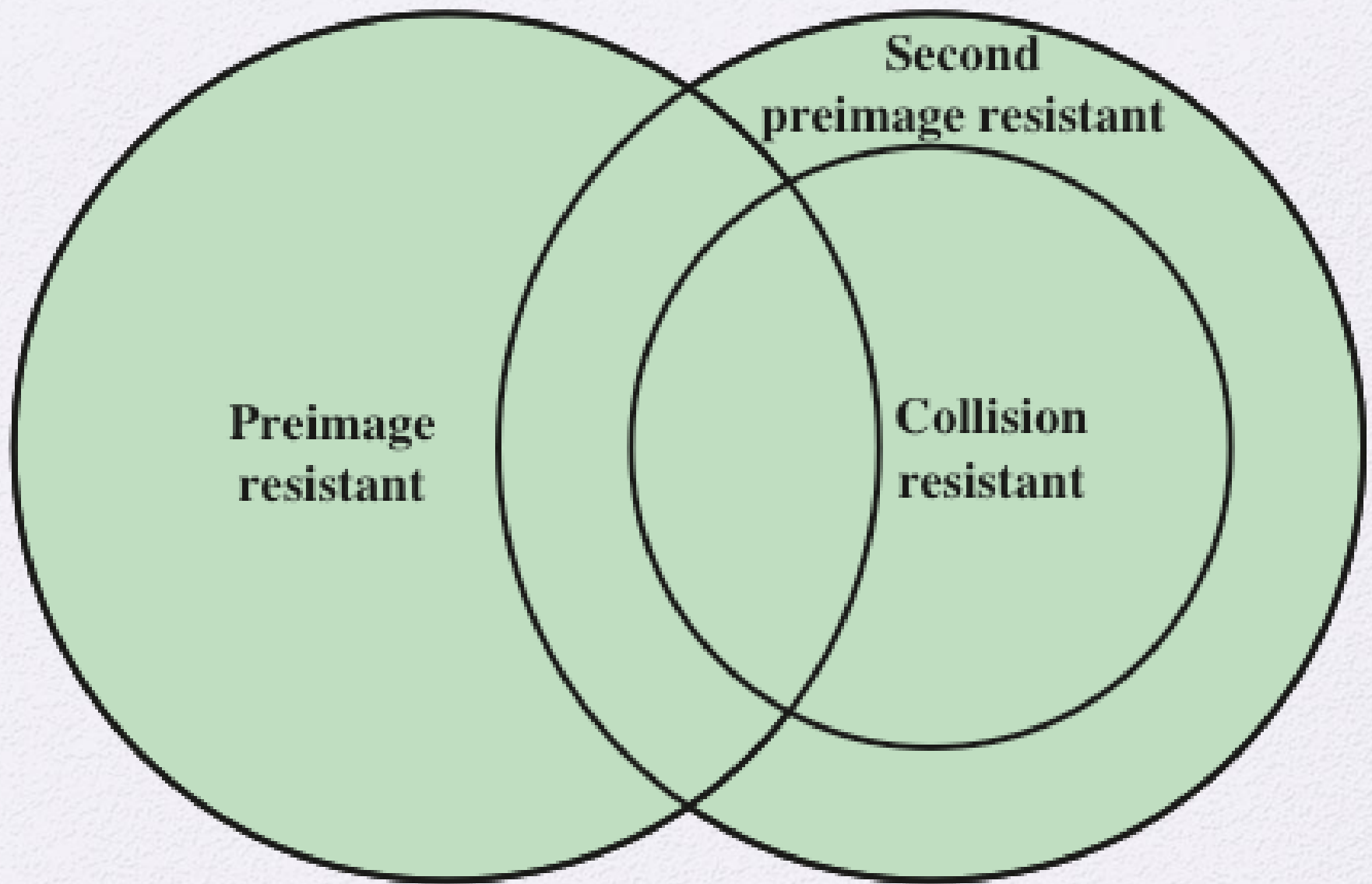


Figure 11.6 Relationship Among Hash Function Properties

Hash Function Resistance Properties Required for Various Data Integrity Applications

	Preimage Resistant	Second Preimage Resistant	Collision Resistant
Hash + digital signature	yes	yes	yes*
Intrusion detection and virus detection		yes	
Hash + symmetric encryption			
One-way password file	yes		
MAC	yes	yes	yes*

* Resistance required if attacker is able to mount a chosen message attack

Attacks on Hash Functions

Brute-Force Attacks

- Does not depend on the specific algorithm, only depends on bit length
- In the case of a hash function, attack depends only on the bit length of the hash value
- Method is to pick values at random and try each one until a collision occurs

Cryptanalysis

- An attack based on weaknesses in a particular cryptographic algorithm
- Seek to exploit some property of the algorithm to perform some attack other than an exhaustive search

Collision resistance and Birthday paradox

- The **birthday paradox** explains why it's easier to find two inputs that produce the same output in a hash function. This is critical in understanding **collision resistance** of hash functions.
- The birthday paradox refers to the fact that:
 - In a group of just 23 people, there's a better than 50% chance that (at least) two of them share the same birthday, despite there being 365 days in a year.
 - For hash functions, the same principle applies: it takes fewer attempts than expected to find a collision.
- **Birthday Paradox Formula:** If there are m possible values, then the number of attempts (k) needed to have a collision probability (p):
$$k = \sqrt{-2 \times m \times \ln(1 - p)} = \sqrt{m} \times \sqrt{\ln\left(\frac{1}{(1-p)^2}\right)}$$
 - For above group of people example:
 - $k = \sqrt{-2 \times m \times \ln(1 - p)} = \sqrt{-2 \times 365 \times \ln(1 - 0.5)} = 22.5 \approx 23$ persons
- **Hash Example:**
 - Estimate the number of attempts (k) needed for a 50% probability of a collision. Assume a hash output of 128-bit.
 - $k = \sqrt{-2 \times m \times \ln(1 - p)} = \sqrt{-2 \times 2^{128} \times \ln(1 - 0.5)} = \sqrt{1.3862 \times 2^{128}} = 1.177 \times 2^{64}$
 - Compare 2^{128} (possible hash output) with 2^{64} (collision attempts)!

A Letter in 2³⁸ Variation

(Letter is located on page 334 in textbook)

As { the } Dean of Blakewell College, I have { had the pleasure of knowing } Cherise
 { -- } known

Rosetti for the { last } four years. She { has been } { a tremendous } { asset to }
 { past } { was } { an outstanding } { role model in }

{ our } school. I { would like to take this opportunity to } recommend Cherise for your
 { the } wholeheartedly

{ school's } graduate program. I { am } { confident } { that } { she } will
 { -- } { feel } { certain } { -- } { Cherise }

{ continue to } succeed in her studies. { She } is a dedicated student and
 { -- } { Cherise }

{ thus far her grades } { have been } { exemplary } . In class, { she }
 { her grades thus far } { are } { excellent } { Cherise }

{ has proven to be } a take-charge { person } { who is } able to successfully develop
 { has been } { individual } { -- } plans and implement them.

{ She } has also assisted { us } in our admissions office. { She } has
 { Cherise } { -- }

{ successfully } demonstrated leadership ability by counseling new and prospective students.
 { -- }

{ Her } advice has been { a great } help to these students, many of whom have
 { Cherise's } { of considerable }

{ taken time to share } their comments with me regarding her pleasant and { encouraging }
 { shared } { reassuring }

attitude. { For these reasons } I { highly recommend } Cherise
 { It is for these reasons that } { offer high recommendations for }

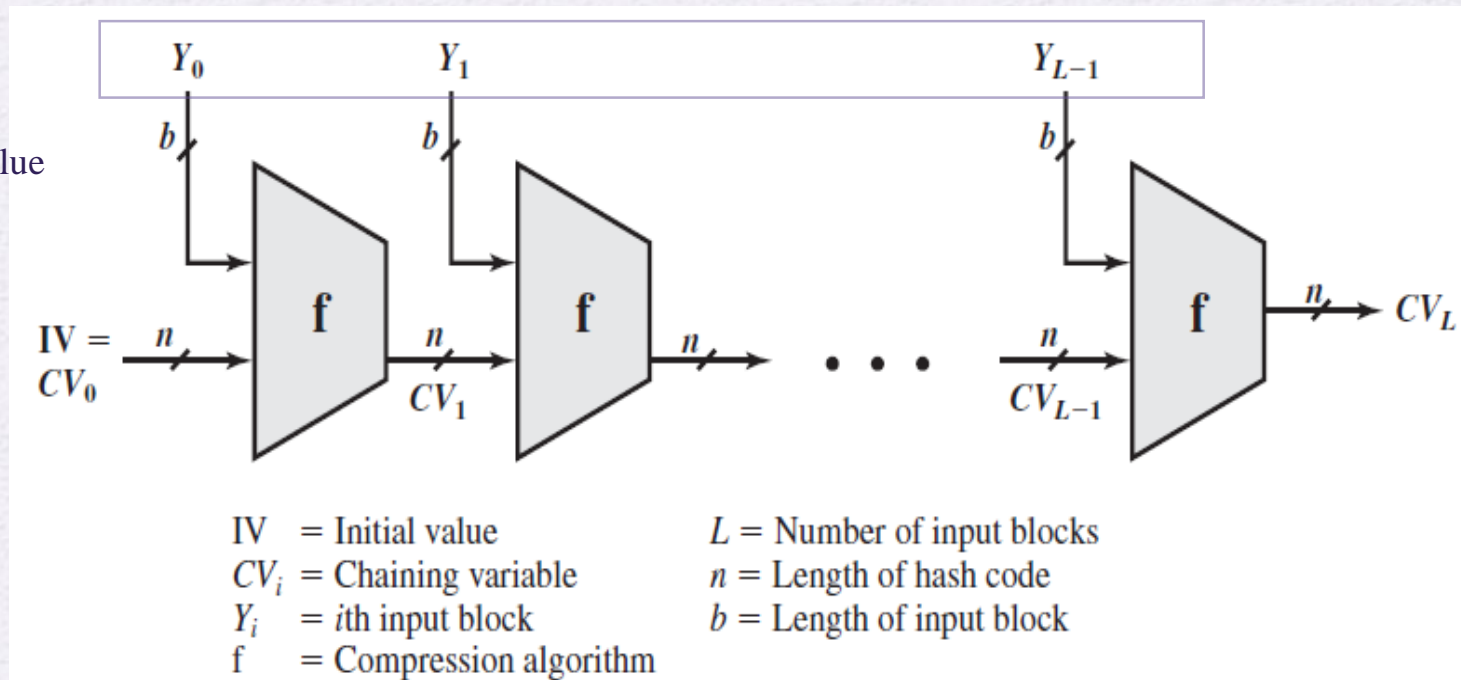
{ without reservation } . Her { ambition } and { abilities } will { truly } be an
 { unreservedly } { drive } { potential } { surely }

{ asset to } your { establishment } .
 { plus for } { school }

Figure 11.7 A Letter in 2³⁸ Variations

General Structure of Hash

- Message size = L blocks
- Block size = b bits
- Message size = $b \times L$
- Typically: $b > n$
- $CV_0 = IV$ = initial n -bit value
- $CV_i = f(CV_{i-1}, Y_{i-1})$
- $H(M) = CV_L$



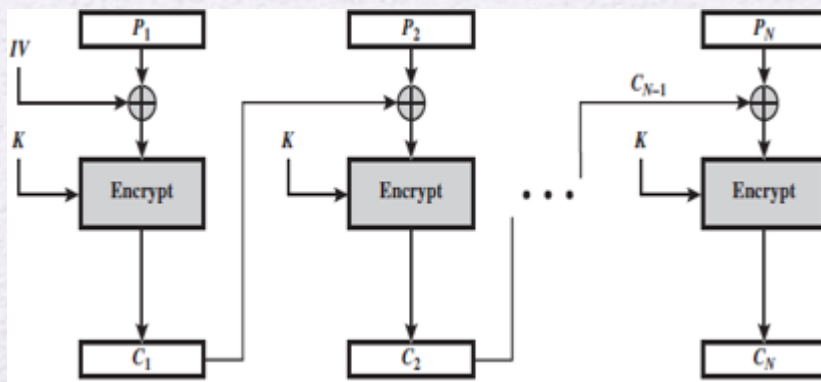
Compression function (f): takes two inputs (n -bit input from the previous step, called the chaining variable, and a b -bit block) and produces an n -bit output.

Hash Function

- Several approaches
 - Symmetric Encryption (Based on Cipher Block Chaining)
 - Secure Hash Algorithm (SHA)

Hash Functions Based on Cipher Block Chaining

- A number of proposals have been made for hash functions based on using a cipher block chaining (CBC) technique, but **without using the secret key**
- One of the first proposals was that of Rabin
 - Divide a message M into fixed-size blocks M_1, M_2, \dots, M_N and use a symmetric encryption system such as DES to compute the hash code G as
$$H_0 = \text{initial value}$$
$$H_i = E(M_i, H_{i-1})$$
$$G = H_N$$
 - Similar to the CBC technique, but in this case, there is **no secret key**
 - As with any hash code, this scheme is subject to the birthday attack
 - If the encryption algorithm is DES and only a 64-bit hash code is produced, the system is vulnerable



Cipher Block Chaining (CBC)

Hash Functions Based on Cipher Block Chaining

- Meet-in-the-middle-attack
 - Another version of the birthday attack used even if the opponent has access to only one message and its valid signature and cannot obtain multiple signings
- It can be shown that some form of **birthday attack** will succeed against any hash scheme involving the use of **cipher block chaining without a secret key**, provided that either the resulting hash code is small enough or that a larger hash code can be decomposed into independent subcodes

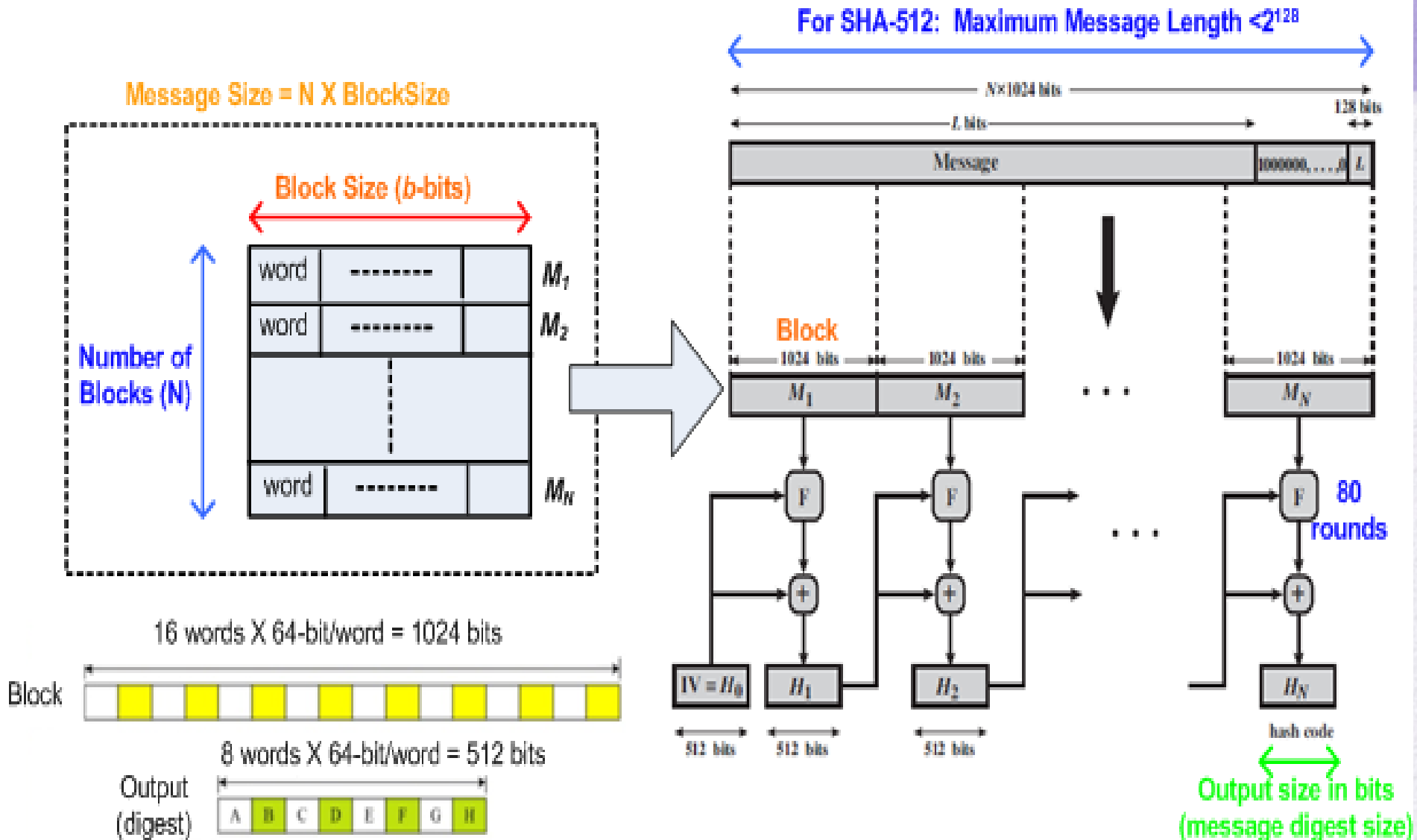
Secure Hash Algorithm (SHA)

- SHA was originally designed by the National Institute of Standards and Technology (NIST) and published as a federal information processing standard (FIPS 180) in 1993
- Has different families: SHA-1, SHA-2 and SHA-3. They vary in some parameters.

Algorithm	Message size	Block size	Word size	Output size (message digest size)	Internal state size (x word size)	Rounds (steps)	Published
SHA-1	$2^{64} - 1$	512	32	160	160 (=5x32)	80	1995
(SHA-2) SHA-256	$2^{64} - 1$	512	32	256	256 (=8x32)	64	2001-2004
(SHA-2) SHA-512	$2^{128} - 1$	1,024	64	512	512 (=8x64)	80	2001-2004
(SHA-3) SHA3-224	Unlimited	1152	64	224	1600 (=5x5x64)	24	2015
(SHA-3) SHA3-256	Unlimited	1088	64	256	1600(=5x5x64)	24	2015
(SHA-3) SHA3-384	Unlimited	832	64	384	1600(=5x5x64)	24	2015
(SHA-3) SHA3-512	Unlimited	576	64	512	1600(=5x5x64)	24	2015
(SHA-3) SHAKE128	d (arbitrary)	1344	64	d	1600(=5x5x64)	24	2015

Will examine this Hash Algorithm First.

SHA Parameters (using SHA-512)



The padded message consists blocks M_1, M_2, \dots, M_N . Each message block M_i consists of 16 64-bit words $M_{i,0}, M_{i,1} \dots M_{i,15}$. All addition is performed modulo 2^{64} .

$$\begin{array}{ll} H_{0,0} = 6A09E667F3BCC908 & H_{0,4} = 510E527FADE682D1 \\ H_{0,1} = BB67AE8584CAA73B & H_{0,5} = 9B05688C2B3E6C1F \\ H_{0,2} = 3C6EF372FE94F82B & H_{0,6} = 1F83D9ABFB41BD6B \\ H_{0,3} = A54FF53A5F1D36F1 & H_{0,7} = 5BE0CDI9137E2179 \end{array}$$

for $i = 1$ **to** N

1. Prepare the message schedule W :

for $t = 0$ **to** 15

$$W_t = M_{i,t}$$

for $t = 16$ **to** 79

$$W_t = \alpha_1^{512}(W_{t-2}) + W_{t-7} + \alpha_5^{512}(W_{t-15}) + W_{t-16}$$

2. Initialize the working variables

$$a = H_{i-1,0} \quad e = H_{i-1,4}$$

$$b = H_{i-1,1} \quad f = H_{i-1,5}$$

$$c = H_{i-1,2} \quad g = H_{i-1,6}$$

$$d = H_{i-1,3} \quad h = H_{i-1,7}$$

3. Perform the main hash computation

for $t = 0$ **to** 79

$$T_1 = h + \text{Ch}(e, f, g) + \left(\sum_1^{512} e \right) + W_t + K_t$$

$$T_2 = \left(\sum_0^{512} a \right) + \text{Maj}(a, b, c)$$

$$h = g$$

$$g = f$$

$$f = e$$

$$e = d + T_1$$

$$d = c$$

$$c = b$$

$$b = a$$

$$a = T_1 + T_2$$

4. Compute the intermediate hash value

$$H_{i,0} = a + H_{i-1,0} \quad H_{i,4} = e + H_{i-1,4}$$

$$H_{i,1} = b + H_{i-1,1} \quad H_{i,5} = f + H_{i-1,5}$$

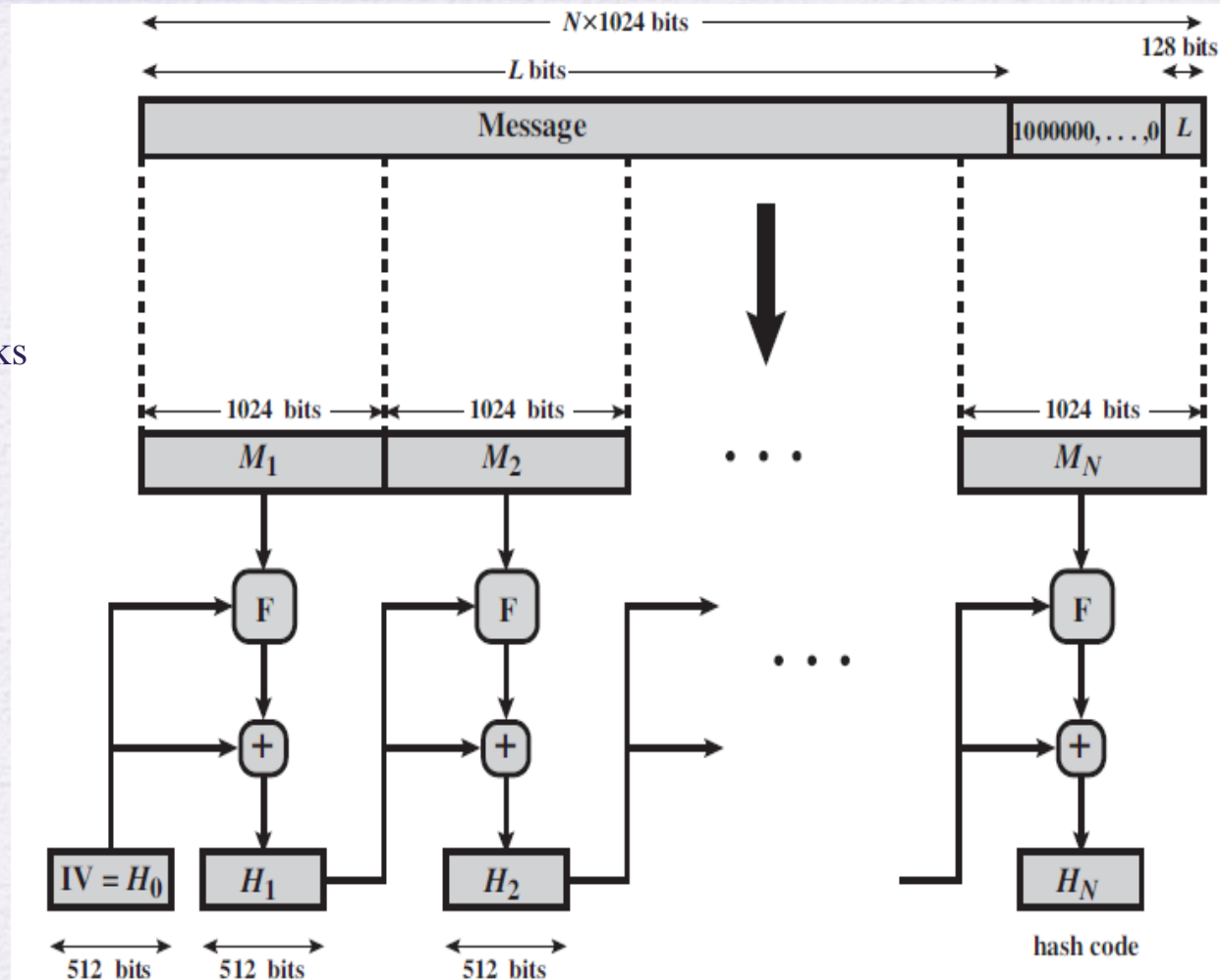
$$H_{i,2} = c + H_{i-1,2} \quad H_{i,6} = g + H_{i-1,6}$$

$$H_{i,3} = d + H_{i-1,3} \quad H_{i,7} = h + H_{i-1,7}$$

return $\{H_{N,0} \parallel H_{N,1} \parallel H_{N,2} \parallel H_{N,3} \parallel H_{N,4} \parallel H_{N,5} \parallel H_{N,6} \parallel H_{N,7}\}$

SHA-512: Steps

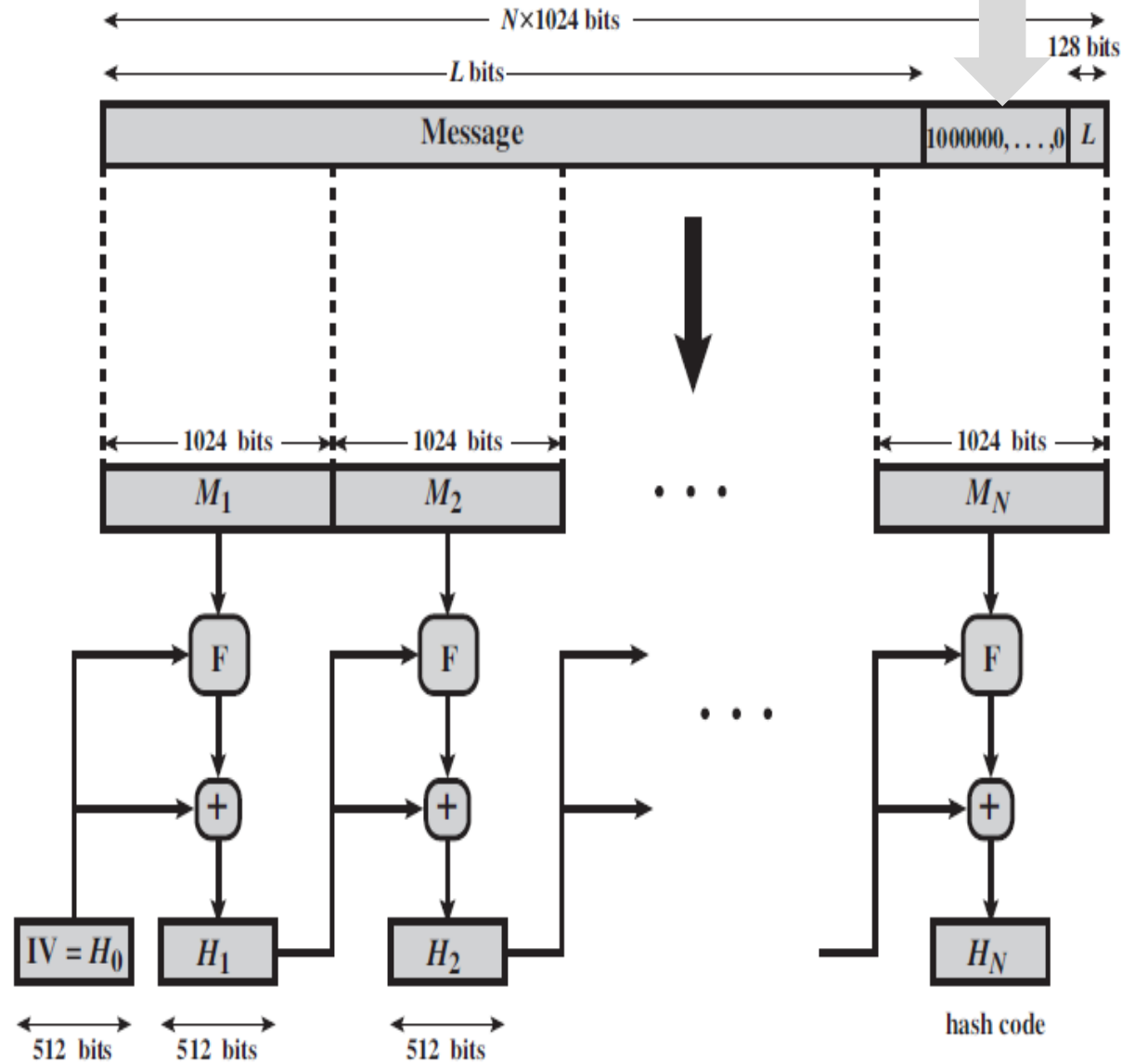
1. Append padding bits
2. Append length
3. Initialize hash buffer
4. Process message in 1024-bit blocks
5. Output



(1) Append Padding bits

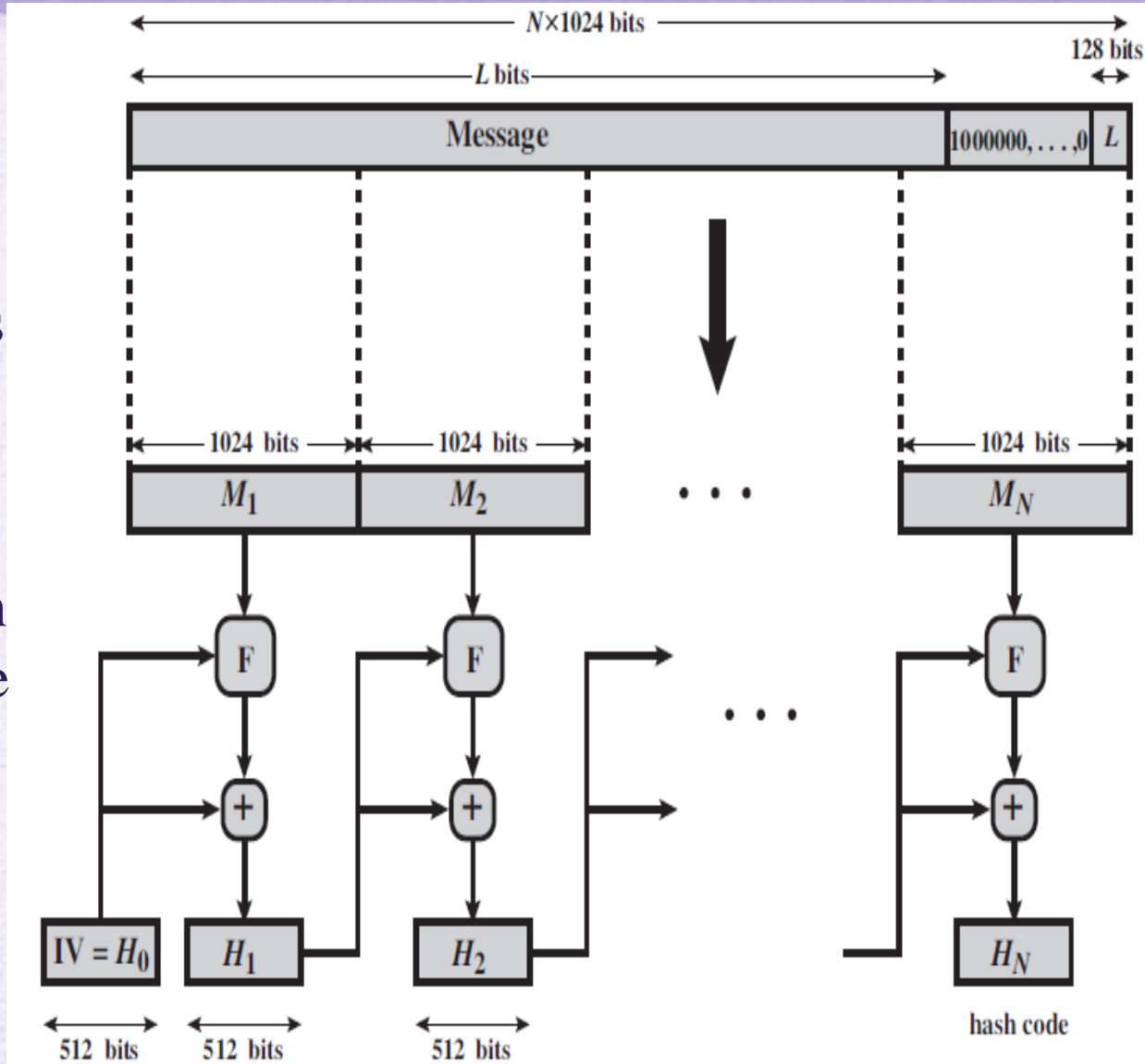
padding

- Message length (L) is expressed in bits.
- The message is padded so that its length (L) is congruent to 896 modulo 1024 .
 - $1024 - 896 = 128$ (which is needed to save L)
 - Message size can be up to $2^{128} - 1$
- Padding is always added, even if the message is already of the desired length.
- The padding consists of a single **1** bit followed by the necessary number of **0** bits.



(2) Append Length

- Block of 128 bits is appended to the message.
- This block is treated as an unsigned 128-bit integer (most significant byte first) and contains the length of the original message (before the padding).



(3) Initialize Hash Buffer for block M_i

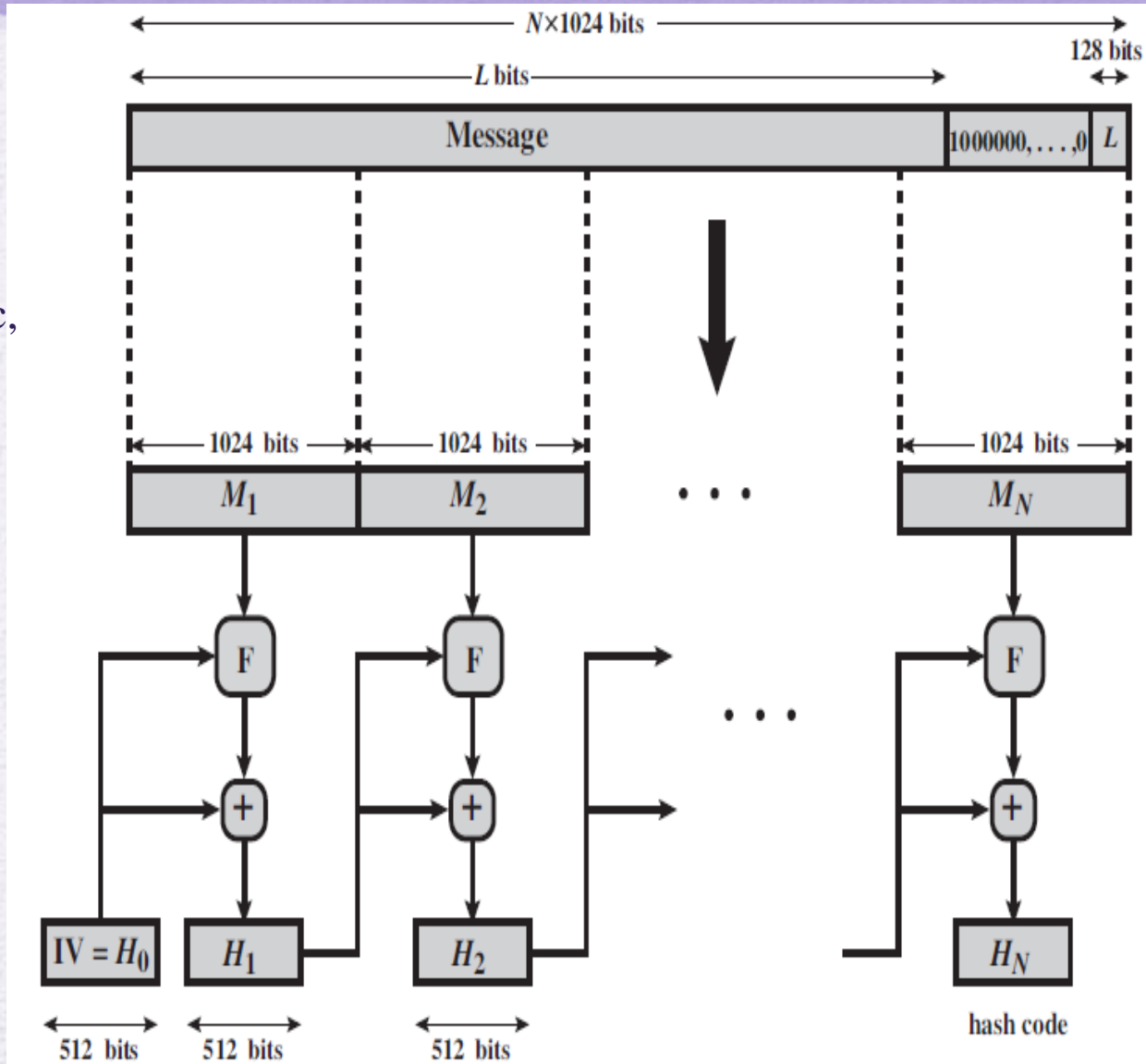
- 512-bit buffer is used to hold intermediate and final results of the hash function.
- The buffer can be represented as eight 64-bit registers (a, b, c, d, e, f, g, h).
- These registers are initialized to the following values:

a = 6A09E667F3BCC908 e = 510E527FADE682D1

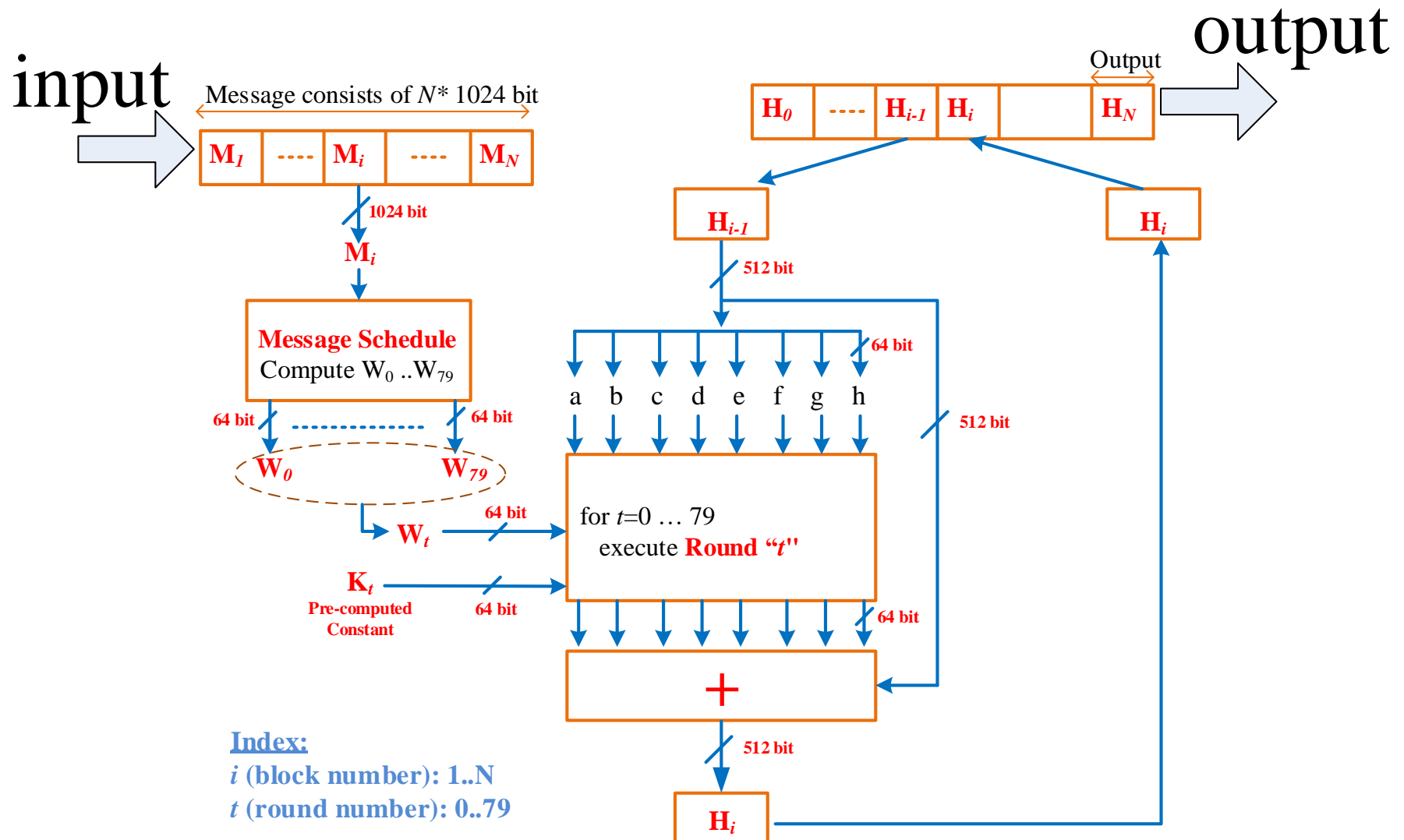
b = BB67AE8584CAA73B f = 9B05688C2B3E6C1F

c = 3C6EF372FE94F82B g = 1F83D9ABFB41BD6B

d = A54FF53A5F1D36F1 h = 5BE0CD19137E2179

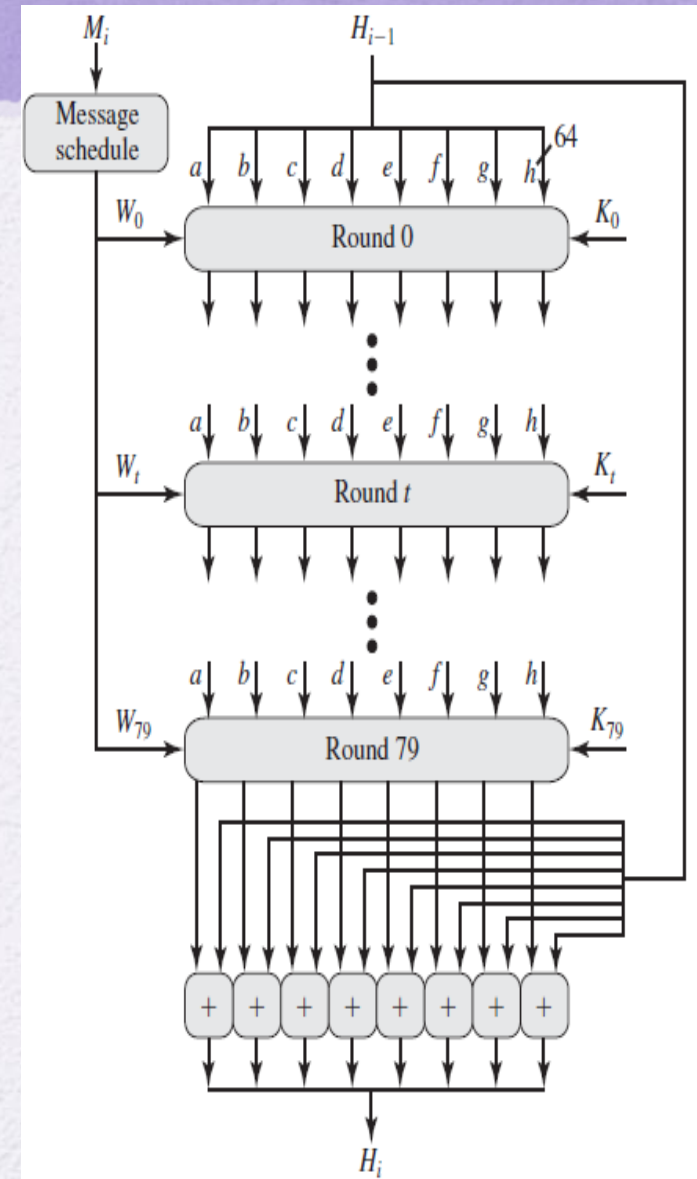


(4) Hash Computation for block M_i



(4) Process blocks in 1024-bit blocks

- Process consists of 80 rounds. The following describes round t , where: $0 \leq t \leq 79$.
- Each round takes as **input** the 512-bit buffer value, $\{a, b, c, d, e, f, g, h\}$, and updates the contents of the buffer.
 - At input to the first round, the buffer has the value of the intermediate hash value, H_{i-1} .
- Each round t makes use of a **64-bit value** W_t , derived from the current 1024-bit block being processed (M_i).
 - W_t values are derived using a message schedule described subsequently.
- Each round also makes use of an additive **64-bit constant** K_t . K_t constants eliminate any regularities in the input data.
- The output of the eightieth round is added to the input to the first round (H_{i-1}) to produce H_i .
 - The addition is done independently for each of the eight words in the buffer with each of the corresponding words in H_{i-1} .



Round Function

$$T_1 = h + \text{Ch}(e, f, g) + \left(\sum_1^{512} e \right) + W_t + K_t$$

$$T_2 = \left(\sum_0^{512} a \right) + \text{Maj}(a, b, c)$$

$$h = g$$

$$g = f$$

$$f = e$$

$$e = d + T_1$$

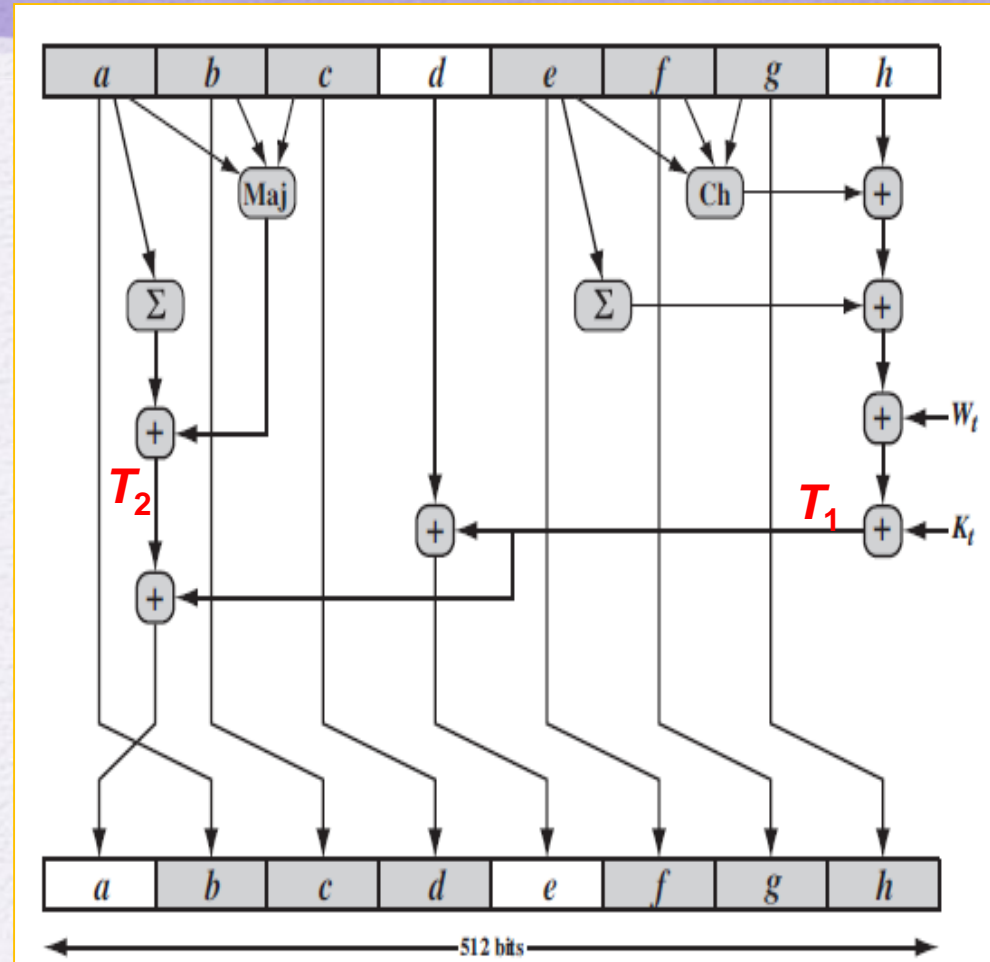
$$d = c$$

$$c = b$$

$$b = a$$

$$a = T_1 + T_2$$

shift
word
to
right



Shift Right by words, except for a and e .

Round Function, cont.

where

t = step number; $0 \leq t \leq 79$

$\text{Ch}(e, f, g) = (e \text{ AND } f) \oplus (\text{NOT } e \text{ AND } g)$
the conditional function: If e then f else g

$\text{Maj}(a, b, c) = (a \text{ AND } b) \oplus (a \text{ AND } c) \oplus (b \text{ AND } c)$
the function is true only if the majority (two or three) of the arguments are true

$\left(\sum_0^{512} a \right) = \text{ROTR}^{28}(a) \oplus \text{ROTR}^{34}(a) \oplus \text{ROTR}^{39}(a)$

$\left(\sum_1^{512} e \right) = \text{ROTR}^{14}(e) \oplus \text{ROTR}^{18}(e) \oplus \text{ROTR}^{41}(e)$

$\text{ROTR}^n(x)$ = circular right shift (rotation) of the 64-bit argument x by n bits

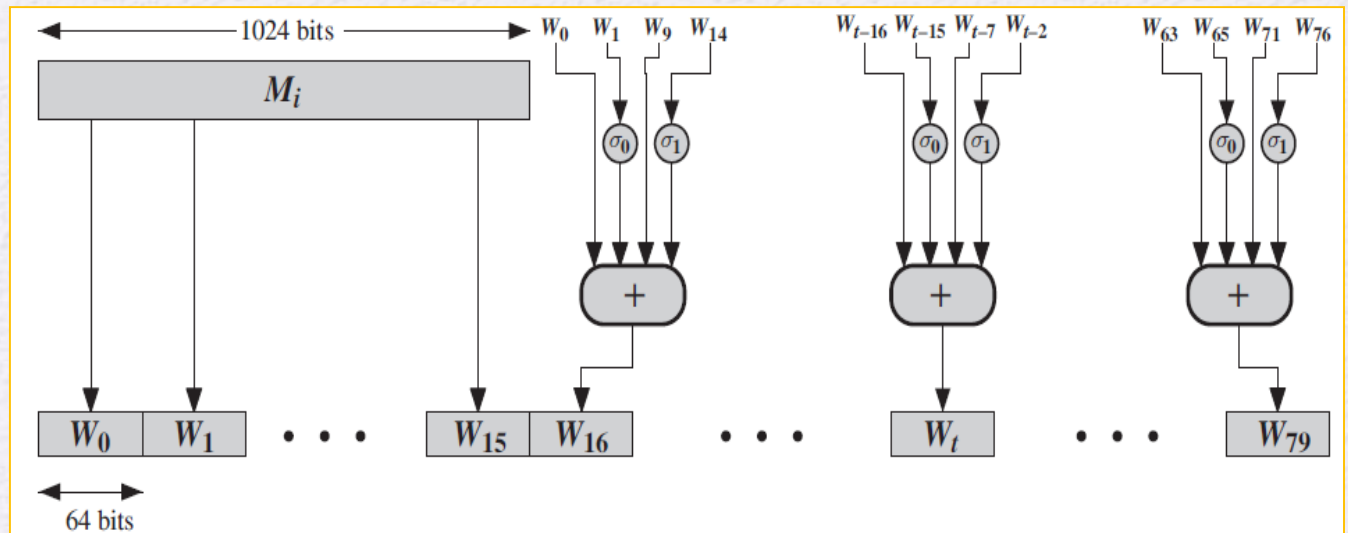
W_t = a 64-bit word derived from the current 1024-bit input block

K_t = a 64-bit additive constant

$+$ = addition modulo 2^{64}

Same function as
 C_{out} in adder

Computing W_t



The first 16 values of are taken directly from the 16 words of the current block.
 The remaining values are defined as:

$$W_t = \sigma_1^{512}(W_{t-2}) + W_{t-7} + \sigma_0^{512}(W_{t-15}) + W_{t-16}$$

where

$$\sigma_0^{512}(x) = \text{ROTR}^1(x) \oplus \text{ROTR}^8(x) \oplus \text{SHR}^7(x)$$

$$\sigma_1^{512}(x) = \text{ROTR}^{19}(x) \oplus \text{ROTR}^{61}(x) \oplus \text{SHR}^6(x)$$

$\text{ROTR}^n(x)$ = circular right shift (rotation) of the 64-bit argument x by n bits

$\text{SHR}^n(x)$ = right shift x by n bits

$+$ = addition modulo 2^{64}

K_t Constants

- Each round also makes use of an additive constant K_t , where $0 \leq t \leq 79$.
- These words represent the first 64 bits of the fractional parts of the cube roots of the first 80 prime numbers.
- The constants provide a “randomized” set of 64-bit patterns, which should eliminate any regularities in the input data.
- Table shows these constants in hexadecimal format (from left to right).

Table 11.4 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	a4506cebd82bde9	bef9a3f7b2c67915	c67178f2e372532b
ca273eceeaa26619c	d186b8c721c0c207	eada7dd6cde0eb1e	f57d4f7fee6ed178
06f067aa72176fba	0a637dc5a2c898a6	113f9804bef90dae	1b710b35131c471b
28db77f523047d84	32caab7b40c72493	3c9ebe0a15c9bebc	431d67c49c100d4c
4cc5d4becb3e42b6	597f299cfc657e2a	5fcb6fab3ad6faec	6c44198c4a475817

K_t Constants Computation

- K_t constants are the first 64 bits of the fractional parts of the cube roots of the first 80 prime numbers. Let us compute K_0 and K_{79}
- The first 80 prime numbers are: 2, 3, 5, ..., 409
- **To compute K_0**
 - 1st prime number = 2
 - $\sqrt[3]{2} = 1.2599210498948731647672106072782$
 - Fraction = 0.2599210498948731647672106072782
 - Convert the fraction part from decimal fraction to 64-bit Hex decimal.
 - $0.2599210498948731647 = \frac{K_0}{2^{64}} = \frac{428a_2f98_d728_ae22_{Hex}}{1_0000_0000_0000_0000_{Hex}} = \frac{4,794,697,086,780,616,226_{Dec}}{18,446,744,073,709,551,616_{Dec}}$
 - $\rightarrow K_0 = 428a_2f98_d728_ae22_{Hex}$
- **To compute K_{79}**
 - 80-th prime number = 409
 - $\sqrt[3]{409} = 7.4229141204362155694375604287751$
 - Fraction = 0.4229141204362155694375604287751
 - Convert the fraction part from decimal fraction to 64-bit Hex decimal.
 - $0.4229141204362155694 = \frac{K_{79}}{2^{64}} = \frac{6c44_198c_4a47_5817_{Hex}}{1_0000_0000_0000_0000_{Hex}} = \frac{7,801,388,544,844,847,127_{Dec}}{18,446,744,073,709,551,616_{Dec}}$
 - $\rightarrow K_{79} = 6c44_198c_4a47_5817_{Hex}$

Initial H hash values: H_0

- The initialize H hash values (i.e. H_0) are the 64-bits fractional parts of the square roots of the first 8 primes: 2, 3, ..19

$$H_{0,0} = 6A09E667F3BCC908$$

$$H_{0,1} = BB67AE8584CAA73B$$

$$H_{0,2} = 3C6EF372FE94F82B$$

$$H_{0,3} = A54FF53A5F1D36F1$$

$$H_{0,4} = 510E527FADE682D1$$

$$H_{0,5} = 9B05688C2B3E6C1F$$

$$H_{0,6} = 1F83D9ABFB41BD6B$$

$$H_{0,7} = 5BE0CDI9137E2179$$

SHA-512: Algorithm

The padded message consists blocks M_1, M_2, \dots, M_N . Each message block M_i consists of 16 64-bit words $M_{i,0}, M_{i,1}, \dots, M_{i,15}$. All addition is performed modulo 2^{64} .

$$\begin{array}{ll} H_{0,0} = 6A09E667F3BCC908 & H_{0,4} = 510E527FADE682D1 \\ H_{0,1} = BB67AE8584CAA73B & H_{0,5} = 9B05688C2B3E6C1F \\ H_{0,2} = 3C6EF372FE94F82B & H_{0,6} = 1F83D9ABFB41BD6B \\ H_{0,3} = A54FF53A5F1D36F1 & H_{0,7} = 5BE0CDI9137E2179 \end{array}$$

for $i = 1$ **to** N

1. Prepare the message schedule W

for $t = 0$ **to** 15

$$W_t = M_{i,t}$$

for $t = 16$ **to** 79

$$W_t = \sigma_1^{512}(W_{t-2}) + W_{t-7} + \sigma_0^{512}(W_{t-15}) + W_{t-16}$$

2. Initialize the working variables

$$\begin{array}{ll} a = H_{i-1,0} & e = H_{i-1,4} \\ b = H_{i-1,1} & f = H_{i-1,5} \\ c = H_{i-1,2} & g = H_{i-1,6} \\ d = H_{i-1,3} & h = H_{i-1,7} \end{array}$$

3. Perform the main hash computation

for $t = 0$ **to** 79

$$T_1 = h + \text{Ch}(e, f, g) + \left(\sum_1^{512} e \right) + W_t + K_t$$

$$T_2 = \left(\sum_0^{512} a \right) + \text{Maj}(a, b, c)$$

$$h = g$$

$$g = f$$

$$f = e$$

$$e = d + T_1$$

$$d = c$$

$$c = b$$

$$b = a$$

$$a = T_1 + T_2$$

4. Compute the intermediate hash value

$$H_{i,0} = a + H_{i-1,0} \quad H_{i,4} = e + H_{i-1,4}$$

$$H_{i,1} = b + H_{i-1,1} \quad H_{i,5} = f + H_{i-1,5}$$

$$H_{i,2} = c + H_{i-1,2} \quad H_{i,6} = g + H_{i-1,6}$$

$$H_{i,3} = d + H_{i-1,3} \quad H_{i,7} = h + H_{i-1,7}$$

return $\{H_{N,0} \parallel H_{N,1} \parallel H_{N,2} \parallel H_{N,3} \parallel H_{N,4} \parallel H_{N,5} \parallel H_{N,6} \parallel H_{N,7}\}$

Example

- We wish to hash a one-block message consisting of three ASCII characters: “abc”, which is equivalent to the following 24-bit binary string: 01100001 01100010 01100011

Recall from step 1 of the SHA algorithm, that the message is padded to a length congruent to 896 modulo 1024. In this case of a single block, the padding consists of $896 - 24 = 872$ bits, consisting of a “1” bit followed by 871 “0” bits. Then a 128-bit length value is appended to the message, which contains the length of the original message (before the padding). The original length is 24 bits, or a hexadecimal value of 18. Putting this all together, the 1024-bit message block, in

message hexadecimal, is

one block

```
6162638000000000 0000000000000000 0000000000000000 0000000000000000
0000000000000000 0000000000000000 0000000000000000 0000000000000000
0000000000000000 0000000000000000 0000000000000000 0000000000000000
0000000000000000 0000000000000000 0000000000000000 0000000000000018
```

L

W's

This block is assigned to the words W_0, \dots, W_{15} of the message schedule, which appears as follows.

$$W_0 = 6162638000000000$$

$$W_1 = 0000000000000000$$

$$W_2 = 0000000000000000$$

$$W_3 = 0000000000000000$$

$$W_4 = 0000000000000000$$

$$W_{10} = 0000000000000000$$

$$W_{11} = 0000000000000000$$

$$W_{12} = 0000000000000000$$

$$W_5 = 0000000000000000$$

$$W_6 = 0000000000000000$$

$$W_7 = 0000000000000000$$

$$W_8 = 0000000000000000$$

$$W_9 = 0000000000000000$$

$$W_{13} = 0000000000000000$$

$$W_{14} = 0000000000000000$$

$$W_{15} = 0000000000000018$$

Variables: a,b,c,d,e,f,g,h

As indicated in Figure 11.12, the eight 64-bit variables, a through h , are initialized to values $H_{0,0}$ through $H_{0,7}$. The following table shows the initial values of these variables and their values after each of the first two rounds.

a	6a09e667f3bcc908	f6afceb8bcfcddf5	1320f8c9fb872cc0
b	bb67ae8584caa73b	6a09e667f3bcc908	f6afceb8bcfcddf5
c	3c6ef372fe94f82b	bb67ae8584caa73b	6a09e667f3bcc908
d	a54ff53a5f1d36f1	3c6ef372fe94f82b	bb67ae8584caa73b
e	510e527fade682d1	58cb02347ab51f91	c3d4ebfd48650ffa
f	9b05688c2b3e6c1f	510e527fade682d1	58cb02347ab51f91
g	1f83d9abfb41bd6b	9b05688c2b3e6c1f	510e527fade682d1
h	5be0cd19137e2179	1f83d9abfb41bd6b	9b05688c2b3e6c1f

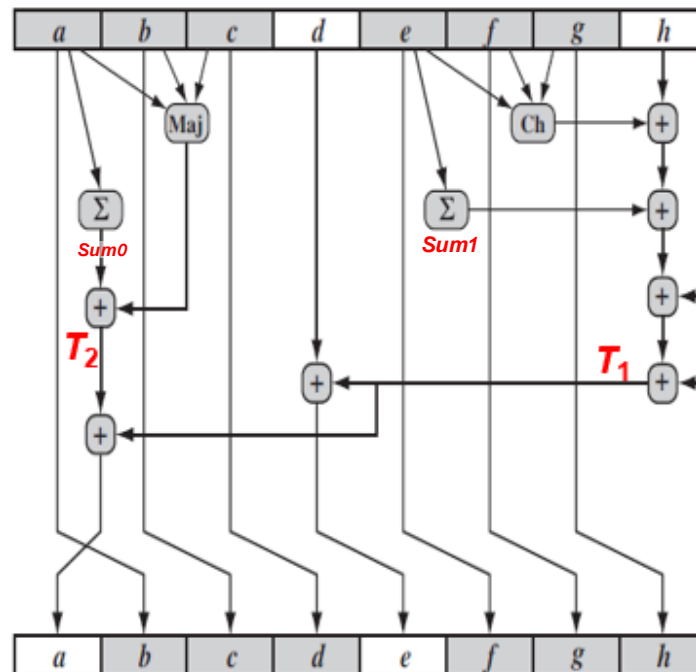
Note that in each of the rounds, six of the variables are copied directly from variables from the preceding round.

Computation for $i=1, t=0$

Computation for : i (block number)=1 t (round number)=0

a=6a09e667f3bcc908
b=bb67ae8584caa73b
c=3c6ef372fe94f82b
d=a54ff53a5f1d36f1

e=510e527fade682d1
f=9b05688c2b3e6c1f
g=1f83d9abfb41bd6b
h=5be0cd19137e2179



T2 Calculations

maj =3a6fe667f69ce92b
sum0=08c4db56aac80c2a
T2 =4334c1bea164f555

$W_0=6162638000000000$

$K_0=428a2f98d728ae22$

T1 Calculations

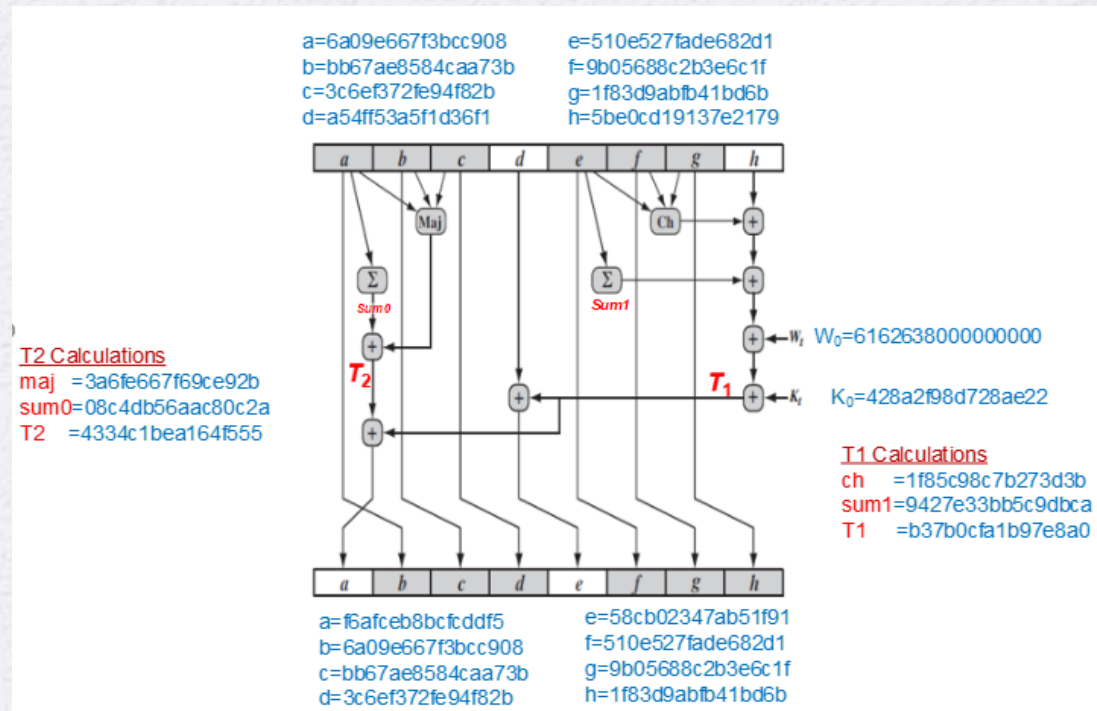
ch =1f85c98c7b273d3b
sum1=9427e33bb5c9dbca
T1 =b37b0cfa1b97e8a0

a=f6afceb8bcfcddf5
b=6a09e667f3bcc908
c=bb67ae8584caa73b
d=3c6ef372fe94f82b

e=58cb02347ab51f91
f=510e527fade682d1
g=9b05688c2b3e6c1f
h=1f83d9abfb41bd6b

Computation for $i=1, t=0$

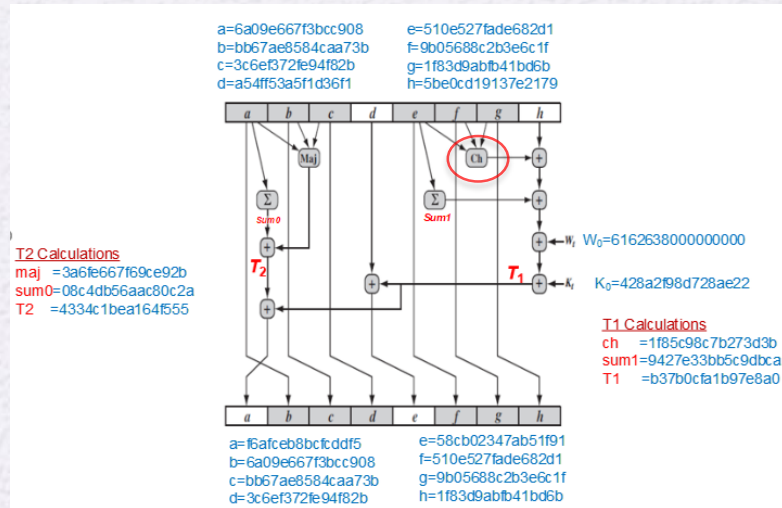
- In the following slides, we present detailed computation for:
 - Block 1: $i=1$
 - First iteration: $t=0$
- We will compute the new values for a, e (the hard ones!)



Computing $\text{Ch}(e, f, g)$

$\text{Ch}(e, f, g) = (e \text{ AND } f) \oplus (\text{NOT } e \text{ AND } g)$
the conditional function: If e then f else g

- e=510e_527f_ade6_82d1
- f=9b05_688c_2b3e_6c1f
- g=1f83_d9ab_fb41_bd6b
- Ch=1f85_c98c_7b27_3d3b
- Bit [63:32], followed by bit [31:00]



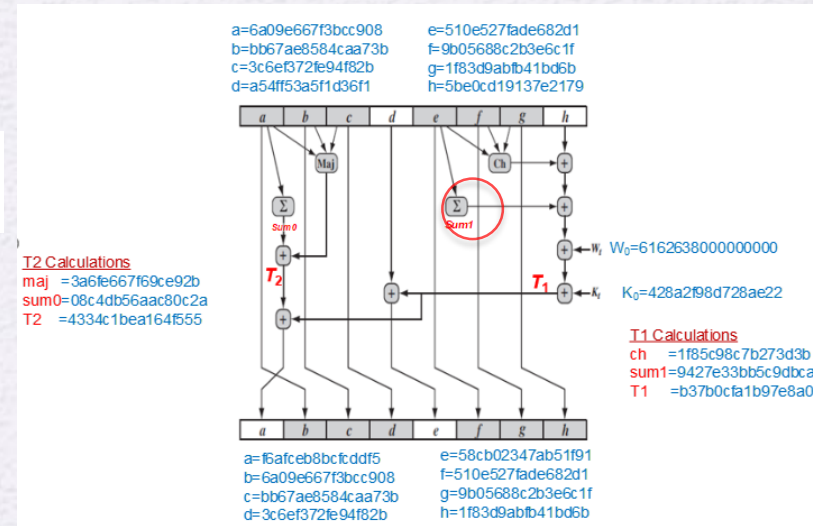
e	0	1	0	1	0	0	0	1	0	0	0	0	1	1	1	0	_	0	1	0	1	0	0	1	0	0	1	1	1	1	1	1	
f	1	0	0	1	1	0	1	1	0	0	0	0	0	1	0	1	_	0	1	1	0	1	0	0	0	1	0	0	0	1	1	0	0
g	0	0	0	1	1	1	1	1	1	0	0	0	0	0	1	1	_	1	1	0	1	1	0	0	1	1	0	1	0	1	0	1	
Ch	0	0	0	1	1	1	1	1	1	0	0	0	0	1	0	1	_	1	1	0	0	1	0	0	1	1	0	0	0	1	1	0	0

e	1	0	1	0	1	1	0	1	1	1	1	0	0	1	1	0	—	1	0	0	0	0	0	1	0	1	1	0	1	0	0	0	1
f	0	0	1	0	1	0	1	1	0	0	1	1	1	1	1	0	—	0	1	1	0	1	1	0	0	0	0	1	1	1	1	1	
g	1	1	1	1	1	0	1	1	0	1	0	0	0	0	0	1	—	1	0	1	1	1	1	0	1	0	1	1	0	1	0	1	
Ch	0	1	1	1	1	0	1	1	0	0	1	0	0	1	1	1	—	0	0	1	1	1	1	0	1	0	0	1	1	1	0	1	1

Computing Sum1

$$\left(\sum_1^{512} e\right) = \text{ROTR}^{14}(e) \oplus \text{ROTR}^{18}(e) \oplus \text{ROTR}^{41}(e)$$

- $e=510e_527f_ade6_82d1$
- $\text{Rotate}^{14}(e)=0b45443949feb79a$
- $\text{Rotate}^{18}(e)=a0b45443949feb79$
- $\text{Rotate}^{41}(e)=3fd6f34168a88729$



$$0b45443949feb79a \oplus a0b45443949feb79 \oplus 3fd6f34168a88729$$

$$\text{Sum1} = 9427e33bb5c9dbca$$

Computing T1

$$T_1 = h + \text{Ch}(e, f, g) + \left(\sum_1^{512} e \right) + W_t + K_t$$

+ = addition modulo 2^{64}

$$h = 5be0cd19137e2179$$

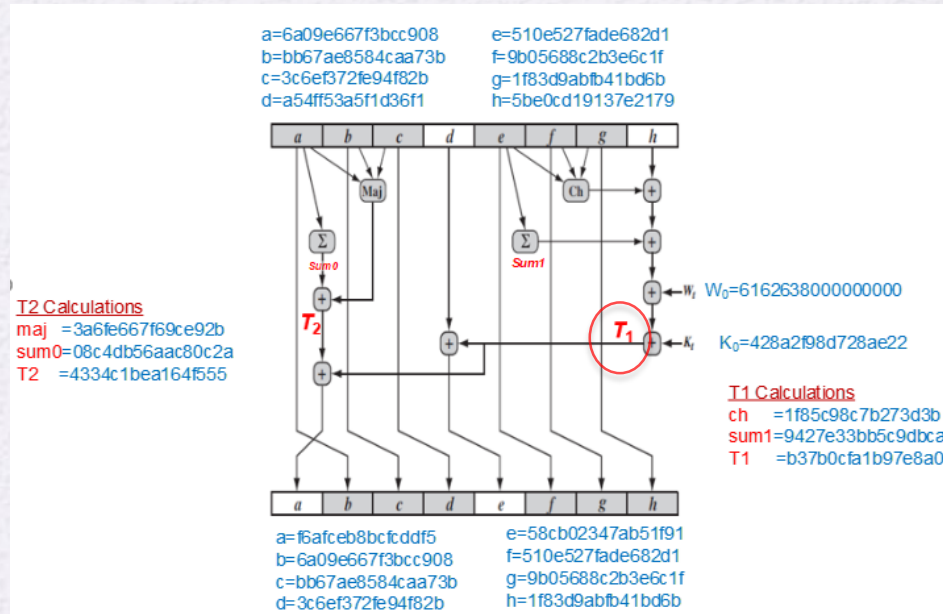
$$\text{Ch} = 1f85c98c7b273d3b$$

$$\text{Sum1} = 9427e33bb5c9dbca$$

$$W_t = 6162638000000000$$

$$K_t = 428a2f98d728ae22$$

$$T_1 = b37b0cfa1b97e8a0$$



Computing Maj(a,b,c)

$$\text{Maj}(a, b, c) = (a \text{ AND } b) \oplus (a \text{ AND } c) \oplus (b \text{ AND } c)$$

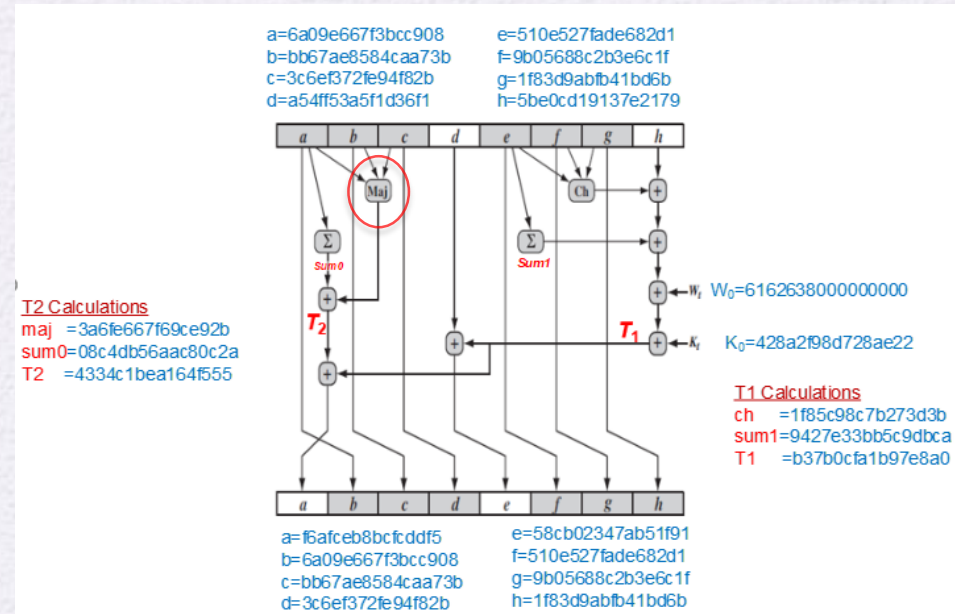
the function is true only if the majority (two or three) of the arguments are true

a = 6a09 e667 f3bc c908

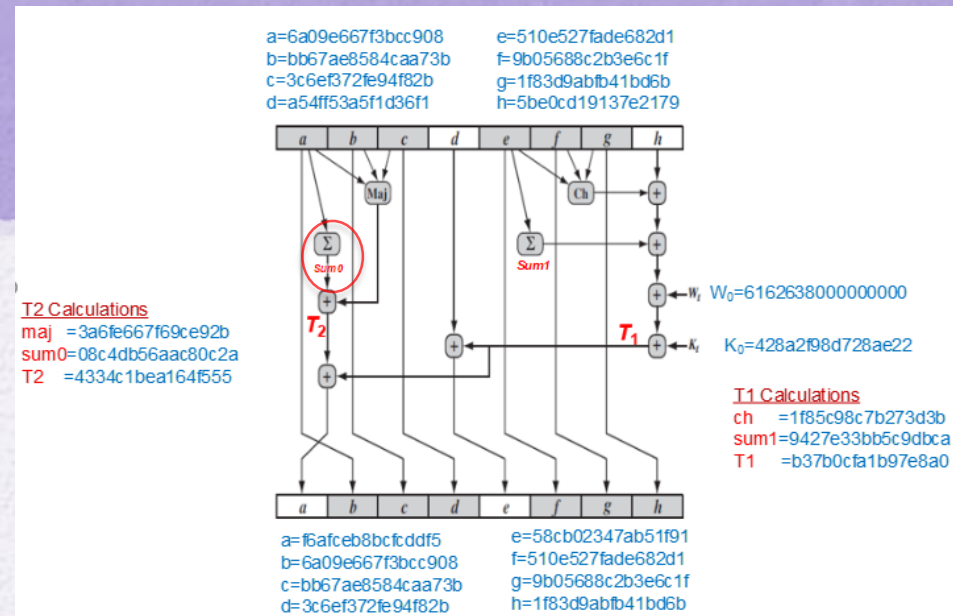
b = bb67 ae85 84ca a73b

c = 3c6e f372 fe94 f82b

Maj = 3a6f e667 f69c e92b



Computing Sum0



$$\left(\sum_0^{512} a \right) = \text{ROTR}^{28}(a) \oplus \text{ROTR}^{34}(a) \oplus \text{ROTR}^{39}(a)$$

- a=6a09_e667_f3bc_c908
- Rotate²⁸(e)= 3bcc9086a09e667f
- Rotate³⁴(e)= fcef32421a827999
- Rotate³⁹(e)= cfe7799210d413cc



$$3bcc9086a09e667f \oplus fcef32421a827999 \oplus cfe7799210d413cc$$

$$\text{Sum0} = 08c4db56aac80c2a$$

Computing T2

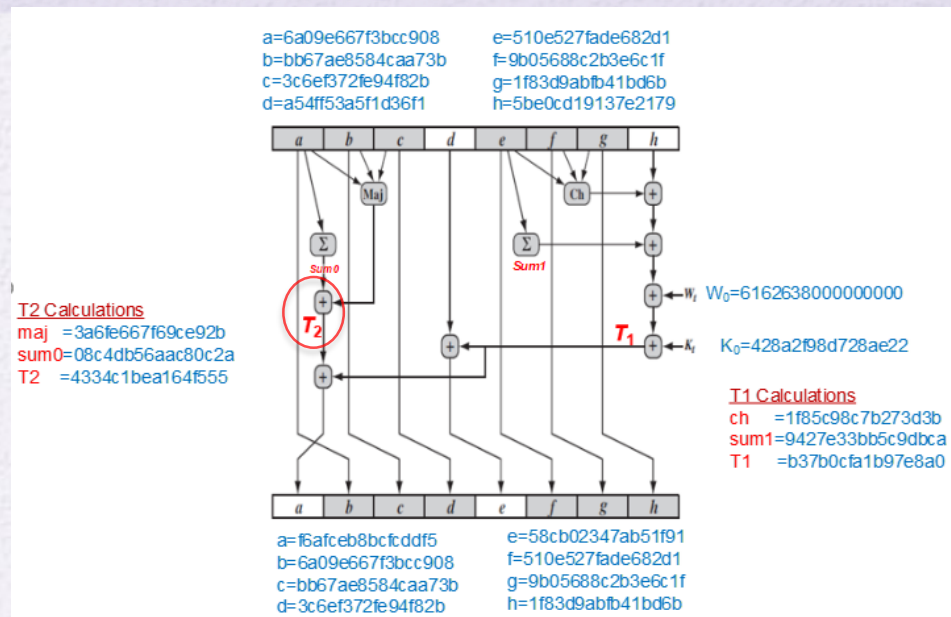
$$T_2 = \left(\sum_0^{512} a \right) + \text{Maj}(a, b, c)$$

+ = addition modulo 2^{64}

Maj = 3a6f e667 f69c e92b

Sum0 = 08c4db56aac80c2a

T2 = 4334c1bea164f555

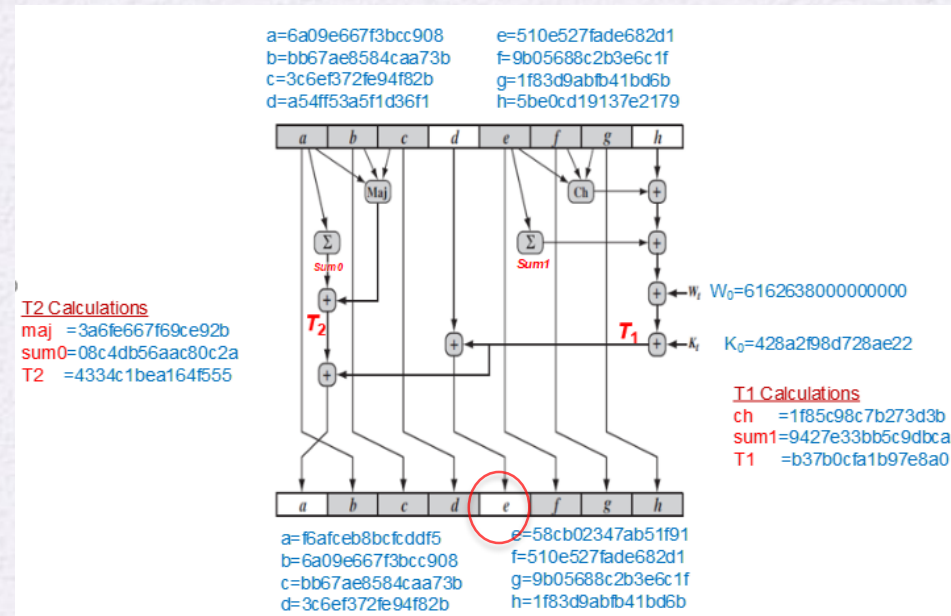


Computing new “e”

$$e = d + T_1$$

+ = addition modulo 2^{64}

$$\begin{array}{rcl} d & = & \text{a54f f53a 5f1d 36f1} \\ T_1 & = & \text{b37b 0cfa 1b97 e8a0} \\ \hline e & = & \text{58cb 0234 7ab5 1f91} \end{array}$$



Computing new “a”

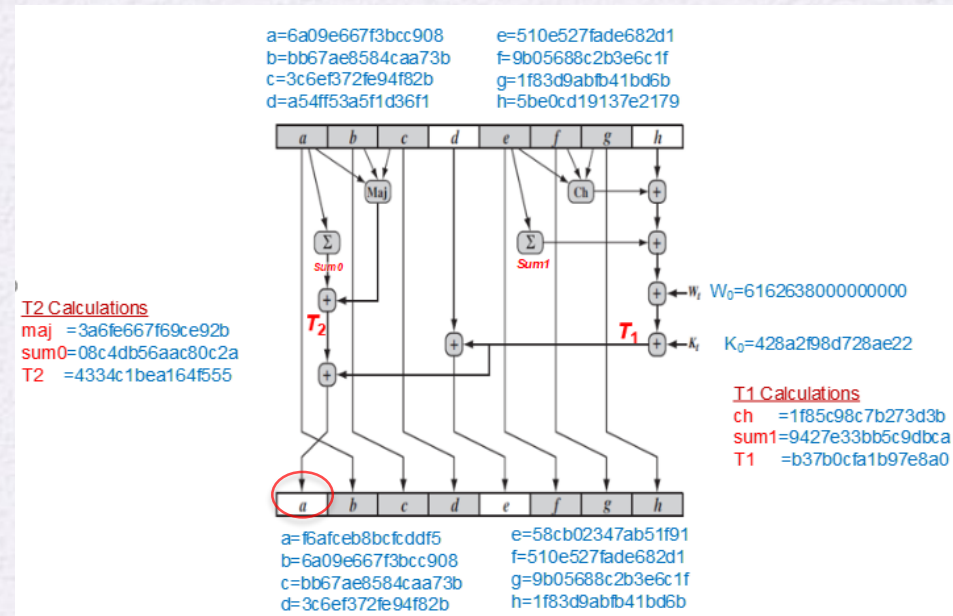
$$a = T_1 + T_2$$

+ = addition modulo 2^{64}

$$T_1 = \text{b37b 0cfa 1b97 e8a0}$$

$$T_2 = \text{4334 c1be a164 f555}$$

$$a = \text{f6af ceb8 bcfc ddf5}$$



Output of Final Round and Hash value

The process continues through 80 rounds. The output of the final round is

73a54f399fa4b1b2 10d9c4c4295599f6 d67806db8b148677 654ef9abec389ca9
d08446aa79693ed7 9bb4d39778c07f9e 25c96a7768fb2aa3 ceb9fc3691ce8326

The hash value is then calculated as

$$H_{1,0} = 6a09e667f3bcc908 + 73a54f399fa4b1b2 = ddaf35a193617aba$$

$$H_{1,1} = bb67ae8584caa73b + 10d9c4c4295599f6 = cc417349ae204131$$

$$H_{1,2} = 3c6ef372fe94f82b + d67806db8b148677 = 12e6fa4e89a97ea2$$

$$H_{1,3} = a54ff53a5f1d36f1 + 654ef9abec389ca9 = 0a9eeee64b55d39a$$

$$H_{1,4} = 510e527fade682d1 + d08446aa79693ed7 = 2192992a274fc1a8$$

$$H_{1,5} = 9b05688c2b3e6c1f + 9bb4d39778c07f9e = 36ba3c23a3feebbd$$

$$H_{1,6} = 1f83d9abfb41bd6b + 25c96a7768fb2aa3 = 454d4423643ce80e$$

$$H_{1,7} = 5be0cd19137e2179 + ceb9fc3691ce8326 = 2a9ac94fa54ca49f$$

The resulting 512-bit message digest is

ddaf35a193617aba cc417349ae204131 12e6fa4e89a97ea2 0a9eeee64b55d39a
2192992a274fc1a8 36ba3c23a3feebbd 454d4423643ce80e 2a9ac94fa54ca49f

SHA-3 (rest is for reading)

SHA-1 has not yet been "broken"

- No one has demonstrated a technique for producing collisions in a practical amount of time
- Considered to be insecure and has been phased out for SHA-2



NIST announced in 2007 a competition for the SHA-3 next generation NIST hash function

- Winning design was announced by NIST in October 2012
- SHA-3 is a cryptographic hash function that is intended to complement SHA-2 as the approved standard for a wide range of applications

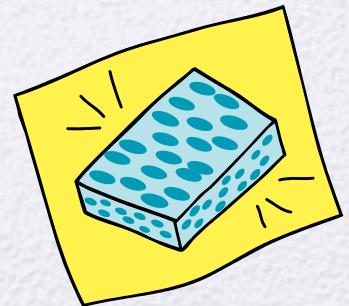
SHA-2 shares the same structure and mathematical operations as its predecessors so this is a cause for concern

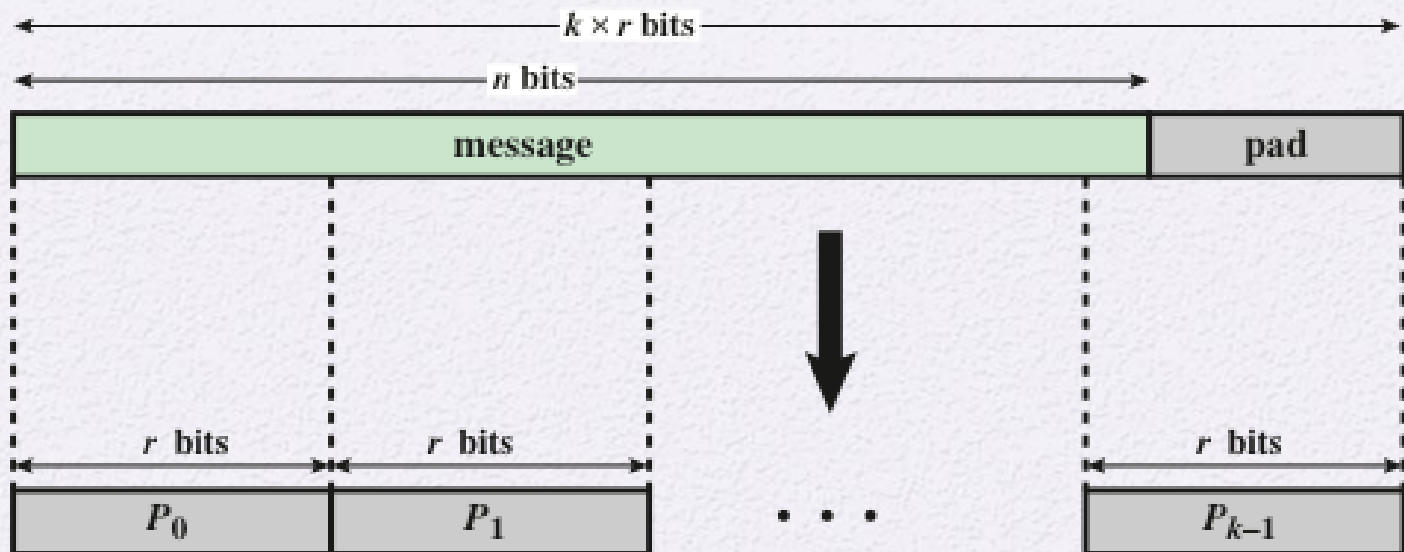
- Because it will take years to find a suitable replacement for SHA-2 should it become vulnerable, NIST decided to begin the process of developing a new hash standard



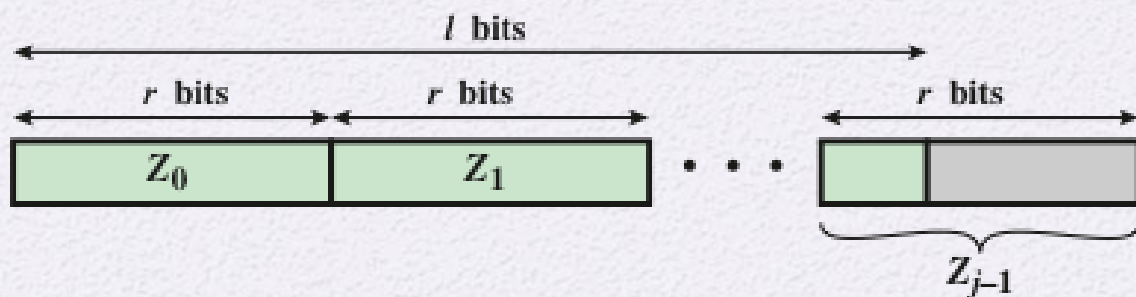
The Sponge Construction

- Underlying structure of SHA-3 is a scheme referred to by its designers as a *sponge construction*
- Takes an input message and partitions it into fixed-size blocks
- Each block is processed in turn with the output of each iteration fed into the next iteration, finally producing an output block
- The sponge function is defined by three parameters:
 - f = the internal function used to process each input block
 - r = the size in bits of the input blocks, called the *bitrate*
 - pad = the padding algorithm





(a) Input



(b) Output

Figure 11.14 Sponge Function Input and Output

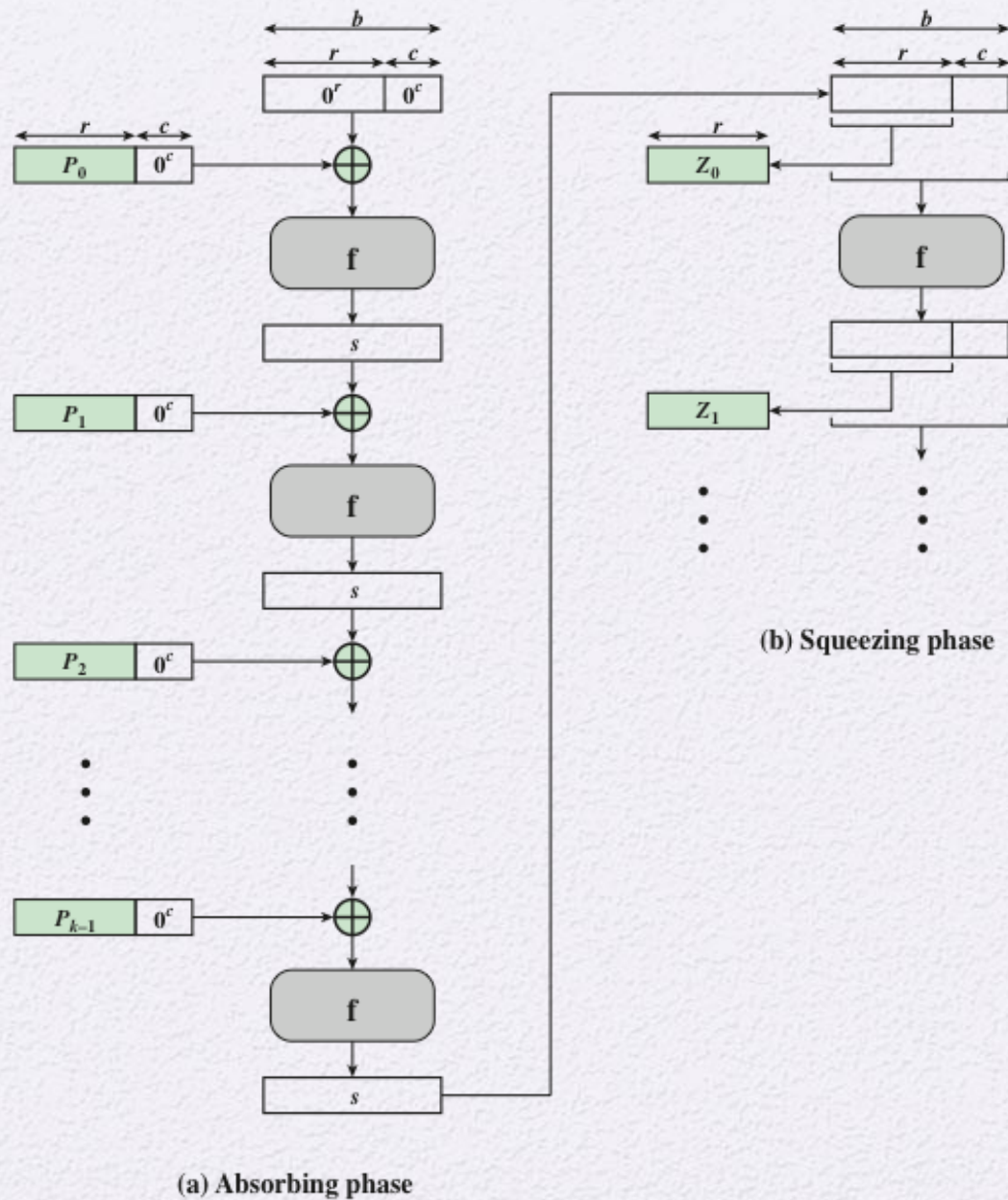


Figure 11.15 Sponge Construction

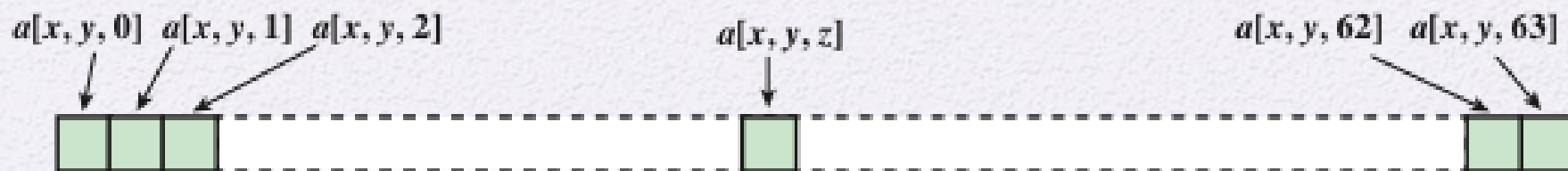
Table 11.5

SHA-3 Parameters

Message Digest Size	224	256	384	512
Message Size	no maximum	no maximum	no maximum	no maximum
Block Size (bitrate r)	1152	1088	832	576
Word Size	64	64	64	64
Number of Rounds	24	24	24	24
Capacity c	448	512	768	1024
Collision resistance	2^{112}	2^{128}	2^{192}	2^{256}
Second preimage resistance	2^{224}	2^{256}	2^{384}	2^{512}

	$x = 0$	$x = 1$	$x = 2$	$x = 3$	$x = 4$
$y = 4$	$L[0, 4]$	$L[1, 4]$	$L[2, 4]$	$L[3, 4]$	$L[4, 4]$
$y = 3$	$L[0, 3]$	$L[1, 3]$	$L[2, 3]$	$L[3, 3]$	$L[4, 3]$
$y = 2$	$L[0, 2]$	$L[1, 2]$	$L[2, 2]$	$L[3, 2]$	$L[4, 2]$
$y = 1$	$L[0, 1]$	$L[1, 1]$	$L[2, 1]$	$L[3, 1]$	$L[4, 1]$
$y = 0$	$L[0, 0]$	$L[1, 0]$	$L[2, 0]$	$L[3, 0]$	$L[4, 0]$

(a) State variable as 5×5 matrix A of 64-bit words



(b) Bit labeling of 64-bit words

Figure 11.16 SHA-3 State Matrix

SHA-3 Iteration Function f

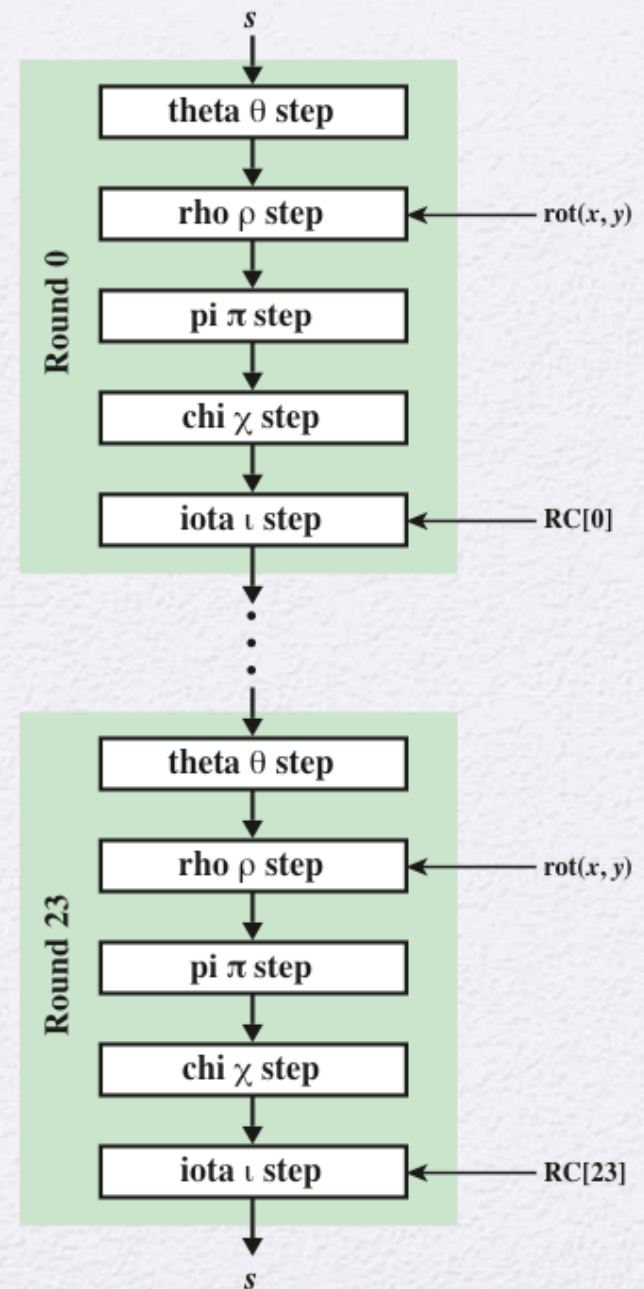
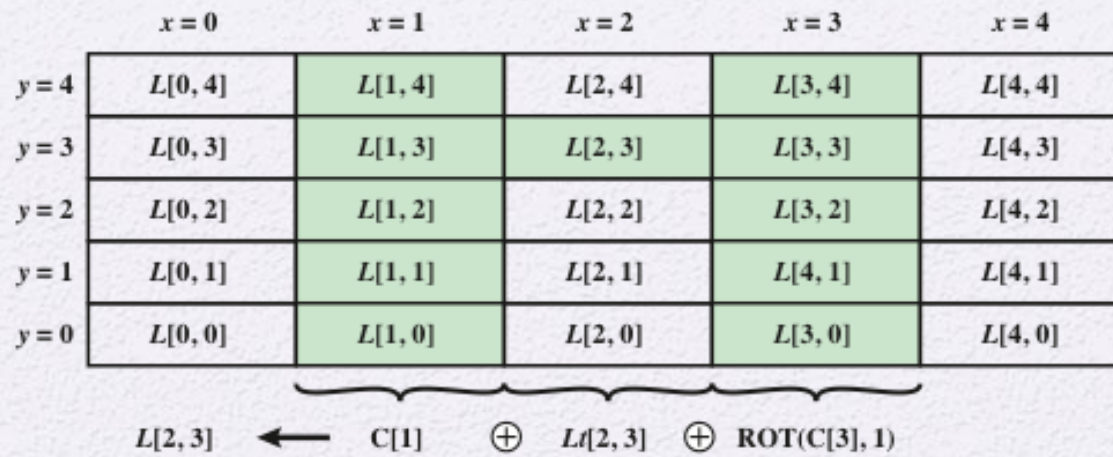


Figure 11.17 SHA-3 Iteration Function f

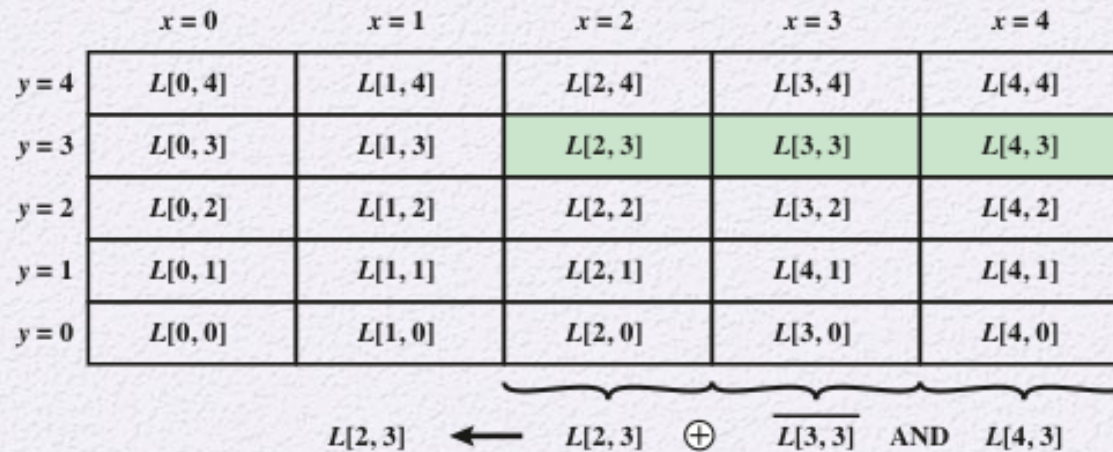
Table 11.6

Step Functions in SHA-3

Function	Type	Description
θ	Substitution	New value of each bit in each word depends its current value and on one bit in each word of preceding column and one bit of each word in succeeding column.
ρ	Permutation	The bits of each word are permuted using a circular bit shift. $W[0, 0]$ is not affected.
π	Permutation	Words are permuted in the 5×5 matrix. $W[0, 0]$ is not affected.
χ	Substitution	New value of each bit in each word depends on its current value and on one bit in next word in the same row and one bit in the second next word in the same row.
ι	Substitution	$W[0, 0]$ is updated by XOR with a round constant.

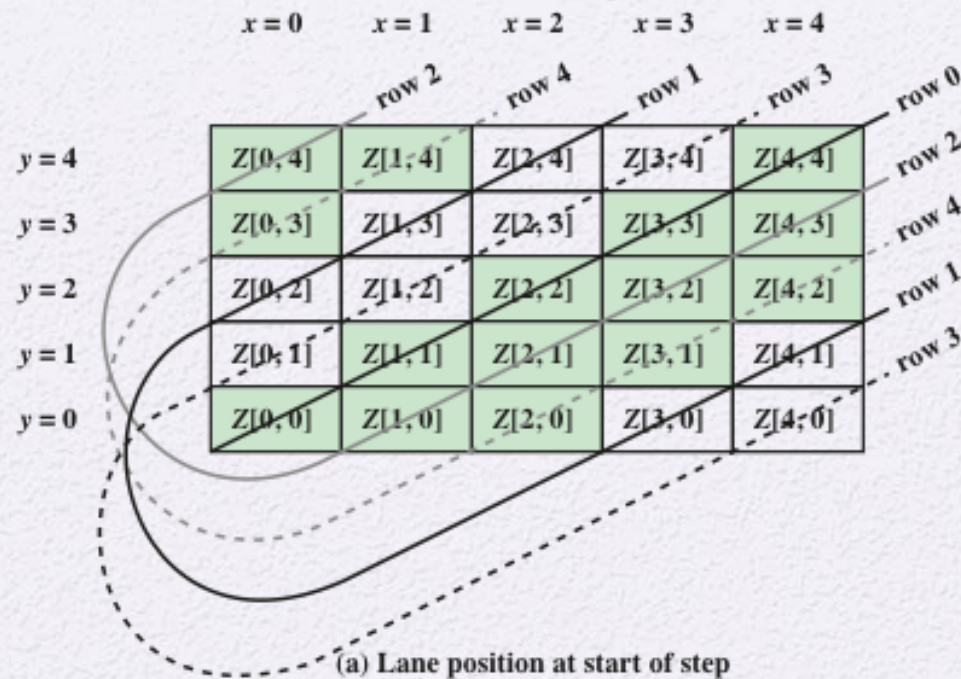


(a) θ step function



(b) χ step function

Figure 11.18 Theta and Chi Step Functions



	$x = 0$	$x = 1$	$x = 2$	$x = 3$	$x = 4$
$y = 4$	$Z[2, 0]$	$Z[3, 1]$	$Z[4, 2]$	$Z[0, 3]$	$Z[1, 4]$
$y = 3$	$Z[4, 0]$	$Z[0, 1]$	$Z[1, 2]$	$Z[2, 3]$	$Z[3, 4]$
$y = 2$	$Z[1, 0]$	$Z[2, 1]$	$Z[3, 2]$	$Z[4, 3]$	$Z[0, 4]$
$y = 1$	$Z[3, 0]$	$Z[4, 1]$	$Z[0, 2]$	$Z[1, 3]$	$Z[2, 4]$
$y = 0$	$Z[0, 0]$	$Z[1, 1]$	$Z[2, 2]$	$Z[3, 3]$	$Z[4, 4]$

(b) Lane position after permutation

Figure 11.19 Pi Step Function

Table 11.8

Round Constants in SHA-3

Round	Constant (hexadecimal)	Number of 1 bits
0	0000000000000001	1
1	0000000000008082	3
2	800000000000808A	5
3	8000000080008000	3
4	000000000000808B	5
5	0000000080000001	2
6	8000000080008081	5
7	8000000000008009	4
8	000000000000008A	3
9	0000000000000088	2
10	0000000080008009	4
11	000000008000000A	3

Round	Constant (hexadecimal)	Number of 1 bits
12	000000008000808B	6
13	800000000000008B	5
14	8000000000008089	5
15	8000000000008003	4
16	8000000000008002	3
17	8000000000000080	2
18	000000000000800A	3
19	800000008000000A	4
20	8000000080008081	5
21	8000000000008080	3
22	0000000080000001	2
23	8000000080008008	4

Summary

- Applications of cryptographic hash functions
 - Message authentication
 - Digital signatures
 - Other applications
- Requirements and security
 - Security requirements for cryptographic hash functions
 - Brute-force attacks
 - Cryptanalysis



- Hash functions based on cipher block chaining
- Secure hash algorithm (SHA)
 - SHA-512 logic
 - SHA-512 round function
- SHA-3
 - The sponge construction
 - The SHA-3 Iteration Function f