Message Authentication using Message Digests and the MD5 Algorithm

Message authentication is important where undetected manipulation of messages can have disastrous effects.

Why Message Authentication?

- Message authentication is important where undetected manipulation of messages can have disastrous effects.
- Examples include Internet Commerce and Network Management.

Message Authentication Using Plain Encryption

(a) Conventional (symmetric) Encryption
$A \rightarrow B: E_K[M]$ •Provides confidentiality —Only A and B share K •Provides a degree of authentication —Could come only from A —Has not been altered in transit —Requires some formatting/redundancy •Does not provide signature —Receiver could forge message —Sender could deny message
(b) Public-Key (asymmetric) Encryption
$A \rightarrow B: E_{KU_b}[M]$ •Provides confidentiality —Only B has KR_b to decrypt •Provides no authentication —Any party could use KU_b to encrypt message and claim to be A
$A \rightarrow B: E_{KR_a}[M]$ •Provides authentication and signature —Only A has KR_a to encrypt —Has not been altered in transit —Requires some formatting/redundancy —Any party can use KU_a to verify signature
$A \rightarrow B: E_{KUb}[E_{KRa}(M)]$ •Provides confidentiality because of KU_b •Provides authentication and signature because of KR_a

Use of Encryption for MAC

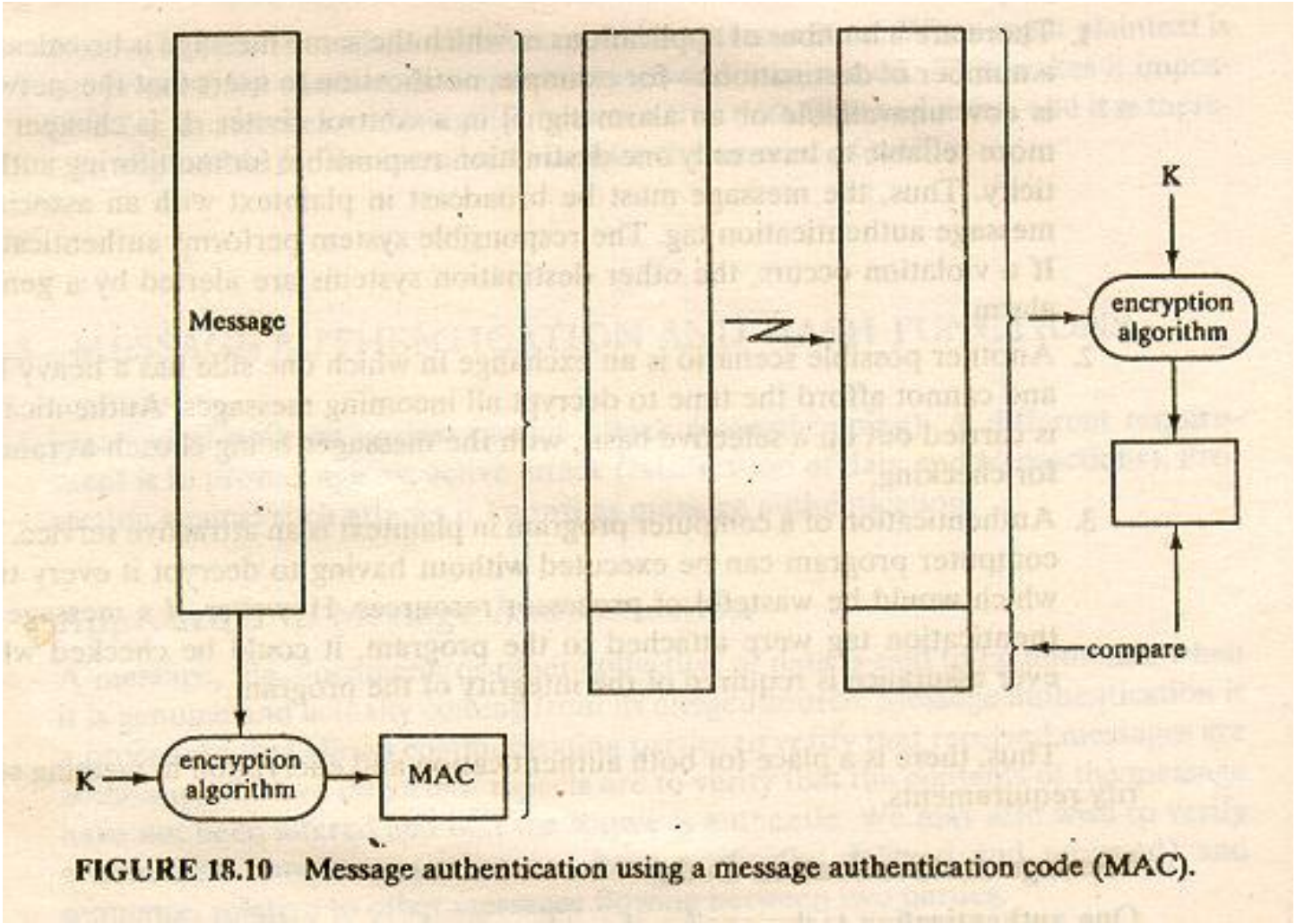
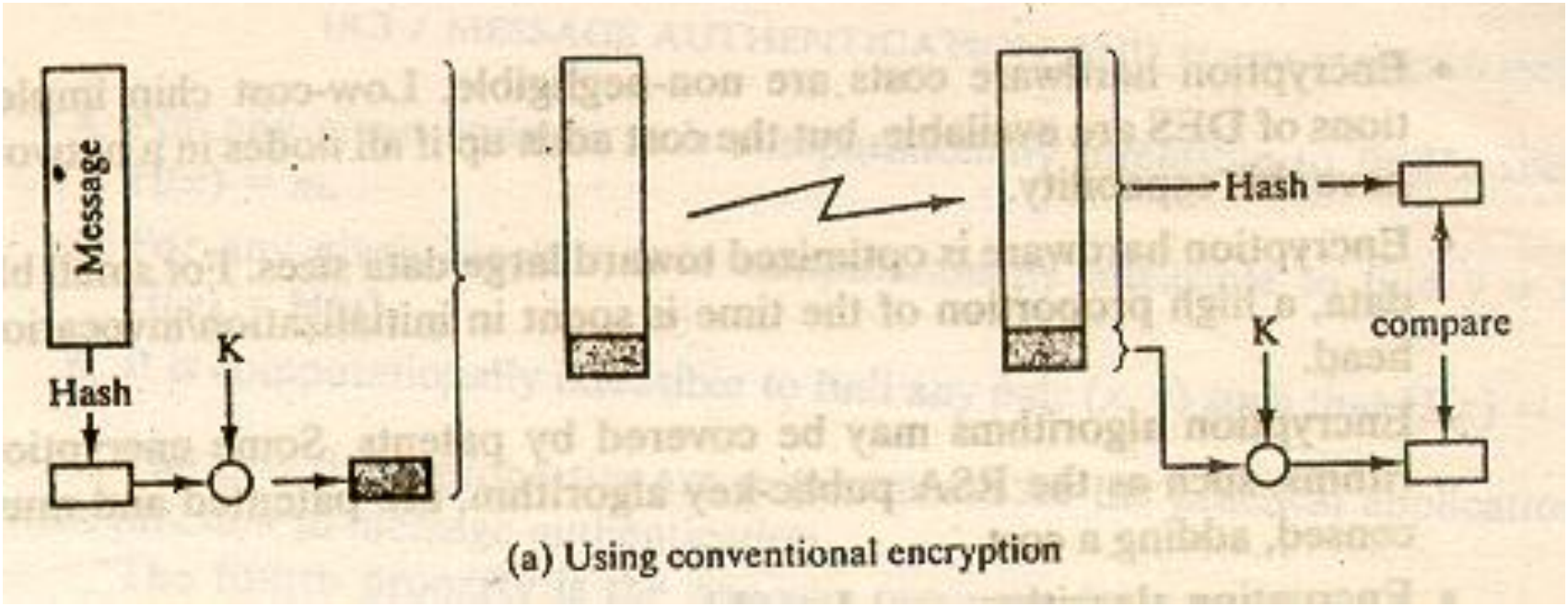


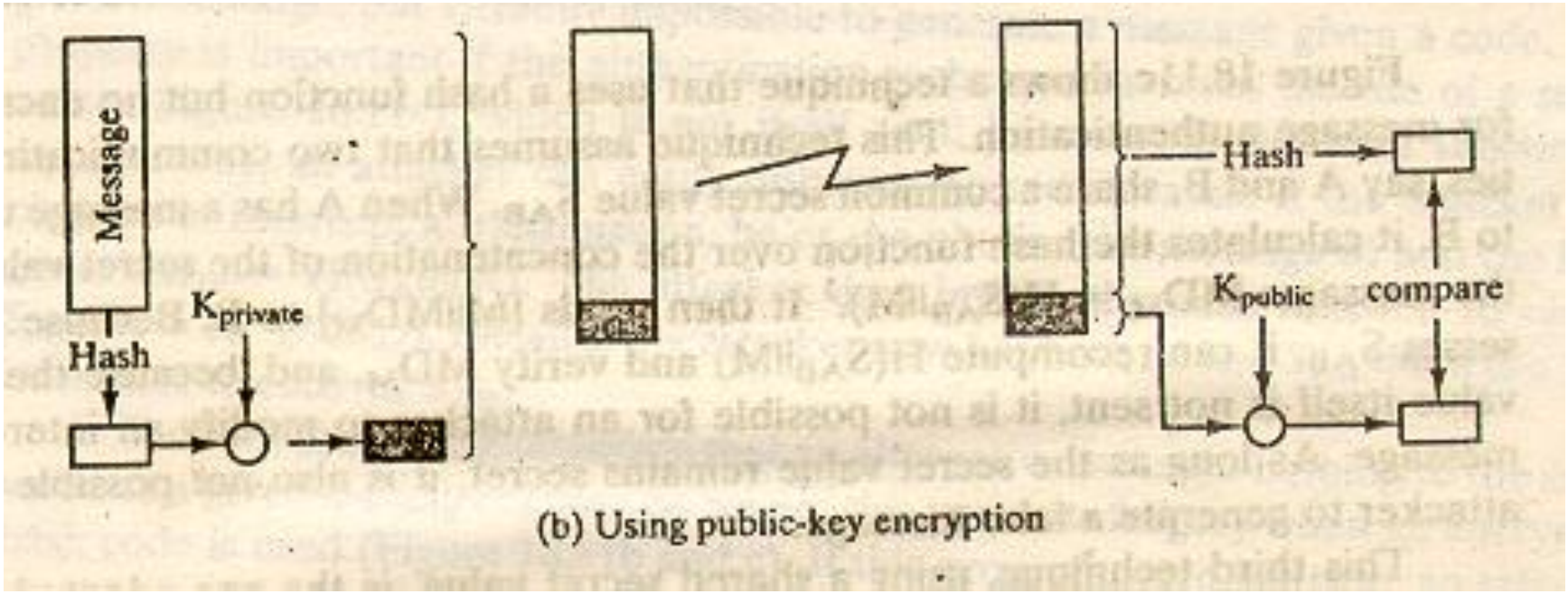
FIGURE 18.10 Message authentication using a message authentication code (MAC).

Message Authentication with Hash Functions using Conventional Encryption



(a) Using conventional encryption

Message Authentication with Hash Functions using Public Key Encryption



(b) Using public-key encryption

What is a hash function?

- A *hash function* H is a transformation that takes an input m and returns a fixed-size string, which is called the hash value h (that is, h = H(m)).
- Hash functions with just this property have a variety of general computational uses, but when employed in cryptography, the hash functions are usually chosen to have some additional properties.

Requirements for Cryptographic Hash Functions

- The basic requirements for a cryptographic hash function are as follows.
  - The input can be of any length.
  - The output has a fixed length.
  - H(x) is relatively easy to compute for any given x.
  - H(x) is one-way.
  - H(x) is collision-free.

H(x) is one-way ...

- A hash function H is said to be *one-way* if it is hard to invert, where ``hard to invert'' means that given a hash value h, it is computationally infeasible to find some input x such that H(x) = h.

Hash Functions as Message Digests

- The hash value represents concisely the longer message or document from which it was computed; this value is called the *message digest*.
- One can think of a message digest as a ``digital fingerprint'' of the larger document.
- Examples of well known hash functions are MD2 and MD5 and SHA

Compression Function

- Damgard and Merkle greatly influenced cryptographic hash function design by defining a hash function in terms of what is called a *compression functio*n.
- A compression function takes a fixed-length input and returns a shorter, fixed-length output.
- Given a compression function, a hash function can be defined by repeated applications of the compression function until the entire message has been processed

Compression Function

- In this process, a message of arbitrary length is broken into blocks whose length depends on the compression function, and "padded" (for security reasons) so the size of the message is a multiple of the block size. The blocks are then processed sequentially, taking as input the result of the hash so far and the current message block, with the final c

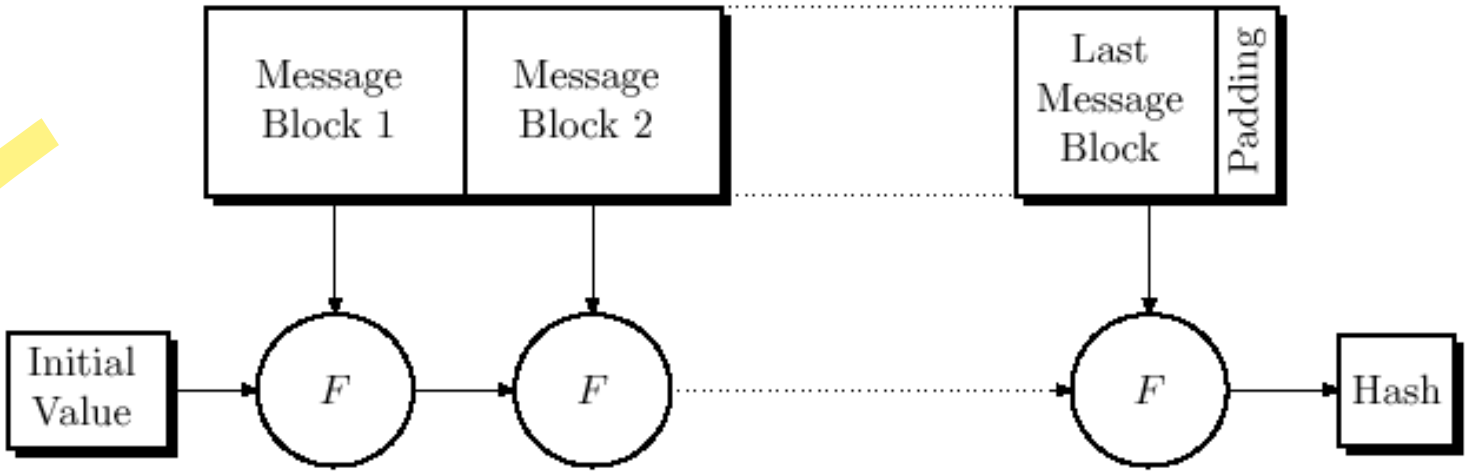


Figure 2.7: *Damgard/Merkle iterative structure for hash functions; F is a compression function.*

MD5 Steps

- The following five steps are performed to compute the message digest of the message.
- Step 1. Append Padding Bits
- Step 2. Append Length
- Step 3. Initialize MD Buffer
- Step 4. Process Message in 16-Word Blocks
- Step 5. Output

Step 1. Append Padding Bits

- The message is "padded" (extended) so that its length (in bits) is congruent to 448, modulo 512. That is, the message is extended so that it is just 64 bits shy of being a multiple of 512 bits long. Padding is always performed, even if the length of the message is already congruent to 448, modulo 512.
- Padding is performed as follows: a single "1" bit is appended to the message, and then "0" bits are appended so that the length in bits of the padded message becomes congruent to 448, modulo 512. In all, at least one bit and at most 512 bits are appended.

Step 2. Append Length

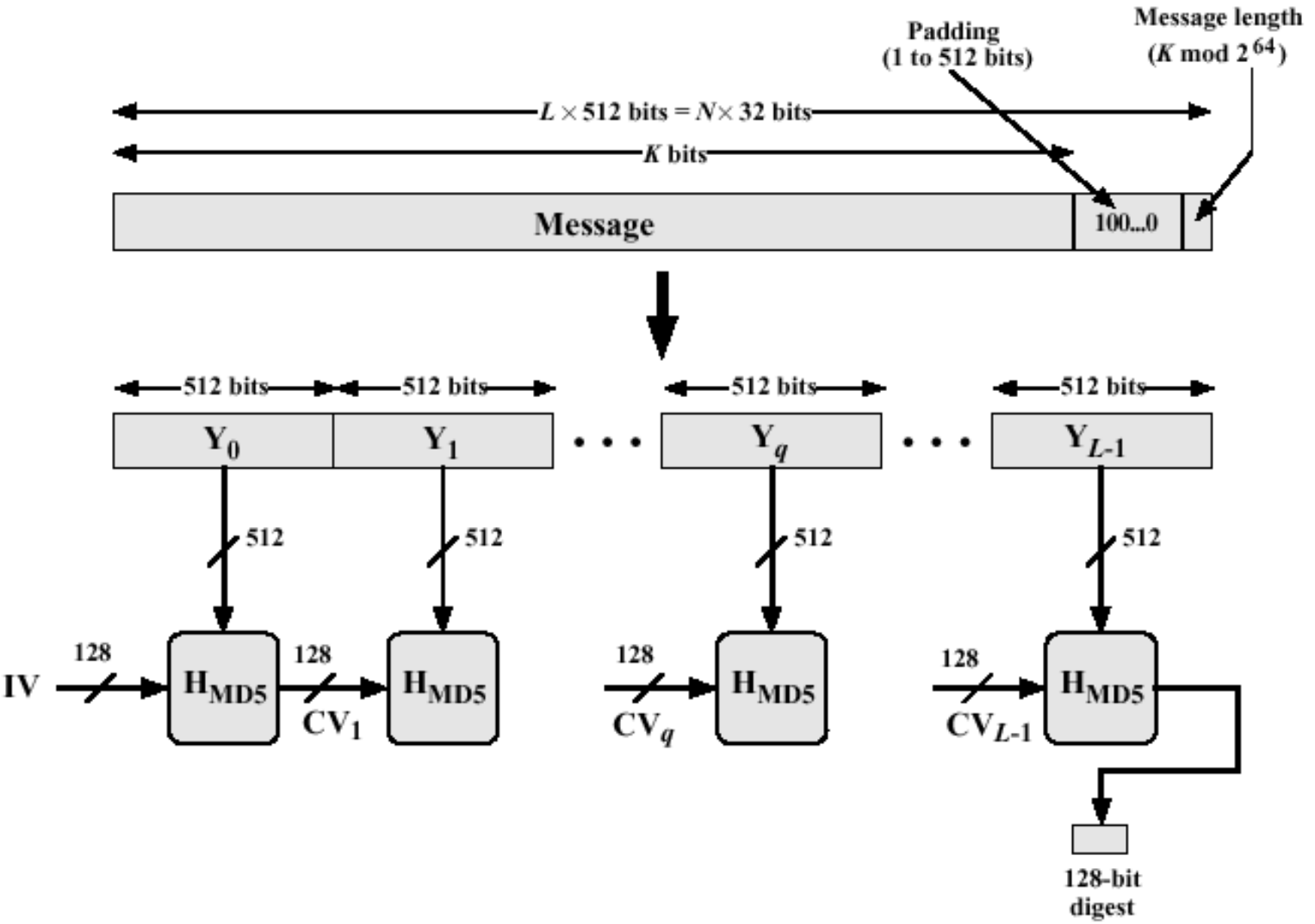
- A 64-bit representation of b (the length of the message before the padding bits were added) is appended to the result of the previous step. In the unlikely event that b is greater than 2^64, then only the low-order 64 bits of b are used. (These bits are appended as two 32-bit words and appended low-order word first in accordance with the previous conventions.)

Step 3. Initialize MD Buffer

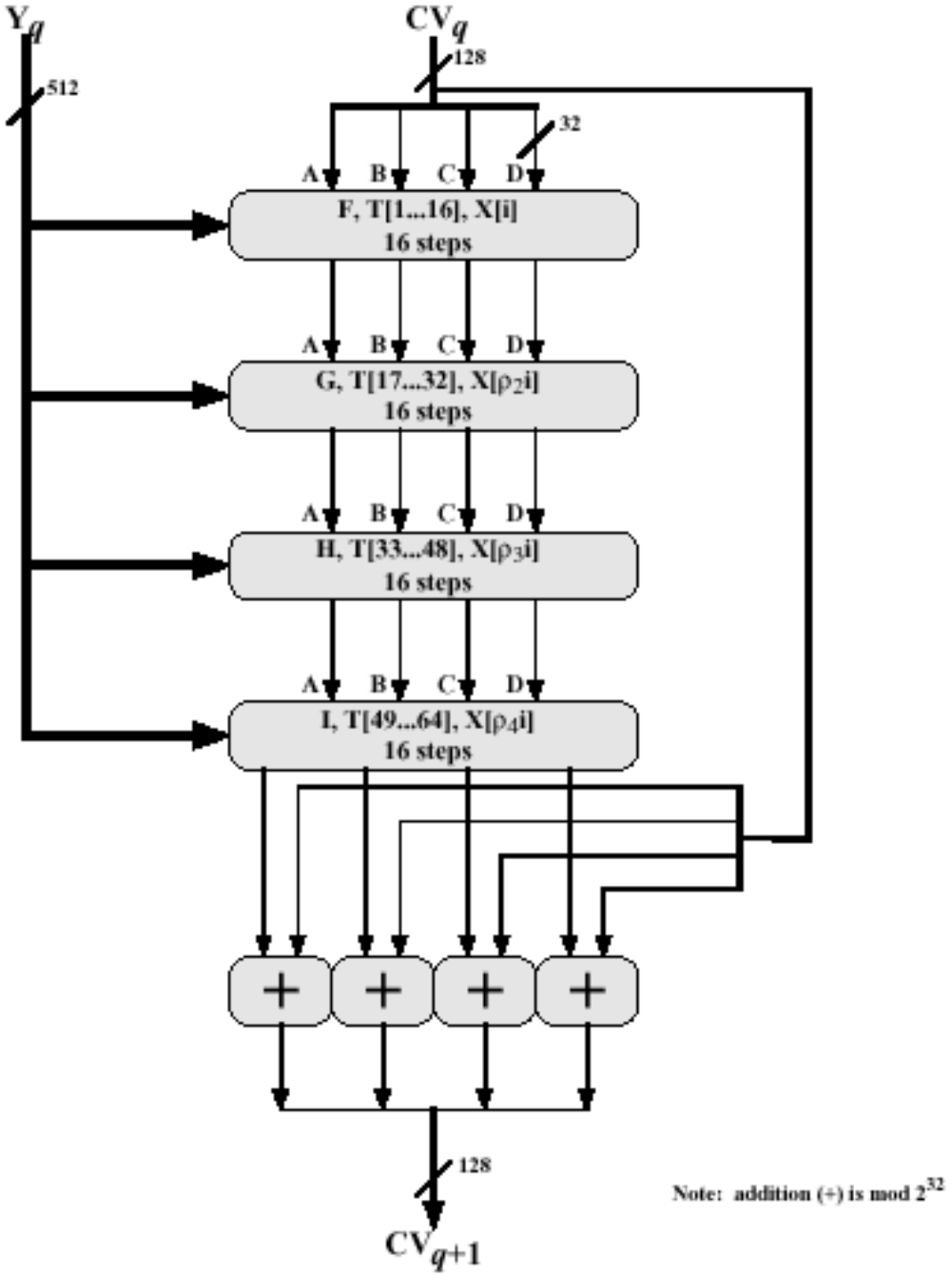
- A four-word buffer (A,B,C,D) is used to compute the message digest.
- Here each of A, B, C, D is a 32-bit register.
- These registers are initialized to the following values in hexadecimal, low-order bytes first):

```
word A: 01 23 45 67
word B: 89 ab cd ef
word C: fe dc ba 98
word D: 76 54 32 10
```

Step 4. Process Message in 16-Word Blocks (4 Rounds)



Step 4. Continued (4 Rounds)



Step 4. Round 1 and 2

```
/* Round 1. */
/* Let [abcd k s i] denote the operation
     a = b + ((a + F(b,c,d) + X[k] + T[i]) <<< s). */
/* Do the following 16 operations. */
[ABCD  0  7  1]  [DABC  1 12  2]  [CDAB  2 17  3]  [BCDA  3 22  4]
[ABCD  4  7  5]  [DABC  5 12  6]  [CDAB  6 17  7]  [BCDA  7 22  8]
[ABCD  8  7  9]  [DABC  9 12 10]  [CDAB 10 17 11]  [BCDA 11 22 12]
[ABCD 12  7 13]  [DABC 13 12 14]  [CDAB 14 17 15]  [BCDA 15 22 16]


/* Round 2. */
/* Let [abcd k s i] denote the operation
     a = b + ((a + G(b,c,d) + X[k] + T[i]) <<< s). */
/* Do the following 16 operations. */
[ABCD  1  5 17]  [DABC  6  9 18]  [CDAB 11 14 19]  [BCDA  0 20 20]
[ABCD  5  5 21]  [DABC 10  9 22]  [CDAB 15 14 23]  [BCDA  4 20 24]
[ABCD  9  5 25]  [DABC 14  9 26]  [CDAB  3 14 27]  [BCDA  8 20 28]
[ABCD 13  5 29]  [DABC  2  9 30]  [CDAB  7 14 31]  [BCDA 12 20 32]
```

Step 4. Round 3 and 4

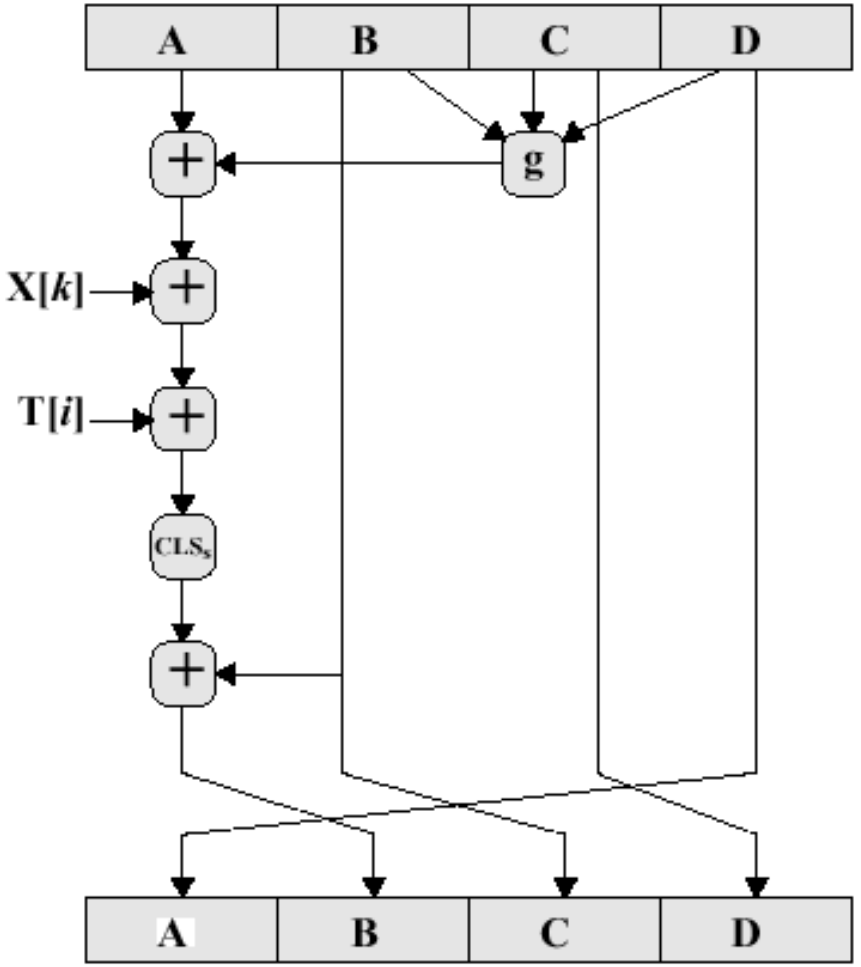
```
/* Round 3. */
/* Let [abcd k s t] denote the operation
     a = b + ((a + H(b,c,d) + X[k] + T[i]) <<< s). */
/* Do the following 16 operations. */
[ABCD  5  4 33]  [DABC  8 11 34]  [CDAB 11 16 35]  [BCDA 14 23 36]
[ABCD  1  4 37]  [DABC  4 11 38]  [CDAB  7 16 39]  [BCDA 10 23 40]
[ABCD 13  4 41]  [DABC  0 11 42]  [CDAB  3 16 43]  [BCDA  6 23 44]
[ABCD  9  4 45]  [DABC 12 11 46]  [CDAB 15 16 47]  [BCDA  2 23 48]


/* Round 4. */
/* Let [abcd k s t] denote the operation
     a = b + ((a + I(b,c,d) + X[k] + T[i]) <<< s). */
/* Do the following 16 operations. */
[ABCD  0  6 49]  [DABC  7 10 50]  [CDAB 14 15 51]  [BCDA  5 21 52]
[ABCD 12  6 53]  [DABC  3 10 54]  [CDAB 10 15 55]  [BCDA  1 21 56]
[ABCD  8  6 57]  [DABC 15 10 58]  [CDAB  6 15 59]  [BCDA 13 21 60]
[ABCD  4  6 61]  [DABC 11 10 62]  [CDAB  2 15 63]  [BCDA  9 21 64]
```

Step 4. Continued

```
/* Then perform the following additions. (That is increment each
   of the four registers by the value it had before this block
   was started.) */
A = A + AA
B = B + BB
C = C + CC
D = D + DD
```

Step 4. Continued



The MD5 Boolean Functions

The functions G, H, and I are similar to the function F, in that they act in "bitwise parallel" to produce their output from the bits of X, Y, and Z, in such a manner that if the corresponding bits of X, Y, and Z are independent and unbiased, then each bit of G(X,Y,Z), H(X,Y,Z), and I(X,Y,Z) will be independent and unbiased. Note that the function H is the bit-wise "xor" or "parity" function of its inputs.

```
F(X,Y,Z) = XY v not(X) Z
G(X,Y,Z) = XZ v Y not(Z)
H(X,Y,Z) = X xor Y xor Z
I(X,Y,Z) = Y xor (X v not(Z))
```

Table T, constructed from the sine function

This step uses a 64-element table T[1 ... 64] constructed from the sine function. Let T[i] denote the i-th element of the table, which is equal to the integer part of 4294967296 times abs(sin(i)), where i is in radians. The elements of the table are given in the following slide.

Table T, constructed from the sine function

T[1] = D76AA478	T[17] = F61E2562	T[33] = FFFA3942	T[49] = F4292244
T[2] = E8C7B756	T[18] = C040B340	T[34] = 8771F681	T[50] = 432AFF97
T[3] = 242070DB	T[19] = 265E5A51	T[35] = 699D6122	T[51] = AB9423A7
T[4] = C1BDCEEE	T[20] = E9B6C7AA	T[36] = FDE5380C	T[52] = FC93A039
T[5] = F57COFAF	T[21] = D62F105D	T[37] = A4BEEA44	T[53] = 655B59C3
T[6] = 4787C62A	T[22] = 02441453	T[38] = 4BDECFA9	T[54] = 8F0CCC92
T[7] = A8304613	T[23] = D8A1E681	T[39] = F6BB4B60	T[55] = FFEFF47D
T[8] = FD469501	T[24] = E7D3FBC8	T[40] = BEBFBC70	T[56] = 85845DD1
T[9] = 698098D8	T[25] = 21E1CDE6	T[41] = 289B7EC6	T[57] = 6FA87E4F
T[10] = 8B44F7AF	T[26] = C33707D6	T[42] = EAA127FA	T[58] = FE2CE6E0
T[11] = FFFF5BB1	T[27] = F4D50D87	T[43] = D4EF3085	T[59] = A3014314
T[12] = 895CD7BE	T[28] = 455A14ED	T[44] = 04881D05	T[60] = 4E0811A1
T[13] = 6B901122	T[29] = A9E3E905	T[45] = D9D4D039	T[61] = F7537E82
T[14] = FD987193	T[30] = FCEFA3F8	T[46] = E6DB99E5	T[62] = BD3AF235
T[15] = A679438E	T[31] = 676F02D9	T[47] = 1FA27CF8	T[63] = 2AD7D2BB
T[16] = 49B40821	T[32] = 8D2A4C8A	T[48] = C4AC5665	T[64] = EB86D391

Step 5. Output

- The message digest produced as output is A, B, C, D.
- That is, we begin with the low-order byte of A, and end with the high-order byte of D.

Other Message Digest Algorithms

- MD4
- SHA-1
- RIPEMD-160

A Comparison of MD5, SHA-1, and RIPEMD-160

	MD5	SHA-1	RIPEMD-160
Digest length	128 bits	160 bits	160 bits
Basic unit of processing	512 bits	512 bits	512 bits
Number of steps	64 (4 rounds of 16)	80 (4 rounds of 20)	160 (5 paired rounds of 16)
Maximum message size	∞	$2^{64} - 1$ bits	$2^{64} - 1$ bits
Primitive logical functions	4	4	5
Additive constants used	64	4	9
Endianness	Little-endian	Big-endian	Little-endian

Relative Performance of Several Hash Functions (coded in C++ on a 266 MHz Pentium)

Algorithm	Mbps
MD5	32.4
SHA-1	14.4
RIPEMD-160	13.6

Note: Coded by Wei Dai; results are posted at http://www.eskimo.com/~weidai/benchmarks.txt