HARDWARE IMPLEMENTATION OF MESSAGE AUTHENTICATION ALGORITHMS FOR INTERNET SECURITY

JANAKA T. DEEPAKUMARA

Faculty of Engineering and Applied Science
Memorial University of Newfoundland

Hardware Implementation of Message Authentication Algorithms for Internet Security

By

Janaka T. Deepakumara ©

A thesis submitted to the
School of Graduate Studies
in partial fulfillment of the
requirements for the degree of
Master of Engineering

May 2002

St. John's Newfoundland Canada

Dedication

To My Dearest Mother, Father and Chamali

who always give me a better life.

Abstract

Verification of integrity and authenticity of information is a prime requirement in computer networks. In open networks such as the Internet, a strong mechanism to provide these security services is essential. With the introduction of Internet Protocol Security (IPSEC), the need has arisen to have a simple, efficient and widely available Message Authentication Code (MAC) mechanism. The standard approach for message authentication in Internet applications has been based on the use of cryptographic hash functions such as Secure Hash Algorithm-1 (SHA-1) and Message Digest 5 (MD5). The wide availability of software implementations, efficiency in software and freedom of license and export restrictions are some of the reasons for adoption of hash-based MACs or HMACs. In high-speed network applications hardware encryption and authentication have become essential to meet the performance requirements. Field Programmable Gate Arrays (FPGAs) are an attractive option because they are capable of providing the required speed, algorithm agility and flexibility of dynamic system evolution. When these factors are considered, FPGA devices are a promising alternative for implementing cryptographic algorithms.

In this research, FPGA implementations of MD5, SHA-1 and HMAC using SHA-1 as the basis hash algorithm have been carried out. MD5 and SHA-1 have been implemented using an iterative and full loop unrolled architectures. The target device has been selected as the XILINX Virtex series FPGA. Performance analysis in terms of hardware utilization and speed are executed. Different design optimization techniques are also discussed.

The Internet is one of the main areas of application of cryptographic hash functions and the size of the message has a considerable impact on the performance of these algorithms. Hence the performance of HMAC both in hardware and software are investigated using four Internet traffic models. The same analysis is performed on CBC-MAC-AES for performance comparison.

Due to the sequential nature of the structure of these algorithms, it is difficult to make them fast enough to ensure suitability for very high-speed applications. Therefore some alternative methods have to be investigated for high-speed applications. One of the proposed algorithms based on universal hashing, the Universal Message Authentication Code (UMAC), is analyzed for its hardware performance. Finally the conclusion and recommendations for future research are presented.

Acknowledgments

I would like to convey my sincere and foremost gratitude to my supervisors as well as my mentors, Dr. Howard M. Heys and Dr. R. Venkatesan for their guidance and financial support throughout this research. Their directions, constructive suggestions and encouragements are not only appreciated, but will also be remembered.

Thanks are also extended to the Dean, faculty of engineering and applied science, Memorial university, for providing the facilities for carrying out this work. I gratefully acknowledge the department of computer science for facilitating Synopsys tools throughout this research. Special thanks are conveyed to Dr. Paul Gillard for his invaluable support for the smooth running of the tools. I gratefully acknowledge Dr. Theodore S. Norvell for providing Xilinx Foundation tools for this research. I also appreciate the timely support provided to me by Mr. Nolan White during problematic periods of the Synopsys tools.

My sincere thanks are also due to all my past and present friends in MUN for giving me a pleasant time throughout this work. Especially the friendly and cordial environment in the CERL was a definite encouragement during my research. Therefore I am grateful to all the professors and friends in the lab. I am greatly beholden to my father, mother, two sisters and brother as well as all the other members of my family for providing me the moral support throughout my life. Last but not least, I pay my heart-felt gratitude to my Chamali for being with me in all the moments during this work. Her unflagging support was a constant source of energy and motivation.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

The significance of information and communications systems for society and the global economy is intensifying with the increasing value and quantity of data that is transmitted and stored on those systems. At the same time those systems and data are also increasingly vulnerable to a variety of threats, such as unauthorized access and use, embezzlement, modification and destruction. As well, the system vulnerability has been increased due to proliferation of computers, increased computer power, interconnectivity, decentralization, growth of networks and number of users and also the convergence of information and communications technologies.

Cryptology is the term which describes the whole meaning of secret communications. This has been originated from the Greek meanings "hidden" and "word". Cryptology can be divided into two subfields: cryptography and cryptanalysis [1]. The cryptographer finds the ways to ensure secrecy and/or authenticity of messages. The cryptanalyst seeks to break that secrecy and/or authenticity by attacking a cipher or by forging coded signals that would be accepted as authentic. Cryptography is an important component of secure communications systems and a variety of applications have been developed that incorporate cryptographic methods to provide data security. Security of information and communications systems involves the assurance of the

confidentiality, integrity, authenticity and availability of those systems and the data that is transmitted and stored on them.

The widespread use of cryptography raises a number of important issues. Governments have many services engaged in the use of cryptography, including protecting the privacy rights of people and organizations, facilitating information and communications systems security, encouraging economic well-being by, in part, promoting electronic commerce, maintaining public safety, enabling the enforcement of laws and the protection of national security, among others. Traditionally, cryptography was most often used by governments. However in recent years cryptography has become an important issue among individuals and businesses as it has become more accessible and more affordable [2]. The explosive growth in computer systems and their interconnections via networks have greatly influenced today's human life. The storing and communicating of information using these systems have become an essential part of our lives. As a result there is a growing awareness of the necessity for information security.

In information security, message authentication and integrity are essential techniques to verify that received messages come from the alleged source and have not been altered during the transit. These techniques may also be useful to verify sequencing, timeliness and to provide non-repudiation. A key element of authentication schemes is the use of a message authentication code (MAC). One technique to produce a MAC is based on using a cryptographic hash function, which is referred to as Hash based Message Authentication Code (HMAC) [3]. The most popular cryptographic hash

functions are the Message Digest 5 (MD5) [3], which was proposed by Ron Rivest, and the Secure Hash Algorithm-1 (SHA-1), which has been certified by the National Institute of Standards and Technology (NIST).

There is a high demand for high quality products and expertise in the field of information security. Recently very high bandwidth networking technologies such as ATM and Gigabit Ethernet are becoming more prevalent. Network applications such as virtual private networks (VPNs) need high-speed cryptographic algorithms to match these new high-speed networks [5].

1.1 Motivation

Internet Protocol Security (IPSEC) [6] is one of the key security standards that provides security services at the IP layer by enabling a system to select required security protocols, determine the algorithm(s) to use for the service(s), and put in place any cryptographic keys required to provide the requested services [6]. IPSEC offers a secure communications across Local Area Networks (LANs), private and public Wide Area Networks (WANs) and the Internet. By employing IPSEC tunnel mode operation, a company can build a secure VPN over the Internet or through a public WAN [7]. IPSEC provides an open framework for implementing industry-standard algorithms. The algorithms employed for MAC value calculation are specified by the security association (SA). Keyed message authentication codes based on symmetric encryption algorithms or one-way hash functions such as MD5 [4] or SHA-1 [8] have both been specified for authentication. IPSEC implementations must support hash based message authentication

codes with MD5 (HMAC-MD5-96) [9] and SHA-1 (HMAC-SHA-1-96) [10]. The algorithm details and the issues in hardware implementation of MD5, SHA-1 and HMAC-SHA-1 are discussed in Chapters 2 and 6 respectively.

In all these applications the performance of IPSEC processing is a crucial issue as cryptographic operations, in general, cause a bottleneck for a processor. In high-speed routers and other networking equipment that apply IPSEC to aggregated traffic, hardware encryption and authentication is almost essential to meet performance objectives [11]. For some applications such equipment may have to handle a large number of security associations and hence key agility and algorithm agility become important issues. There is an increasing interest in high-speed cryptographic accelerators for IPSEC applications such as VPNs. Most of the available products are microprocessor based cryptographic accelerators. They accelerate the computationally intensive algorithms of encryption and authentication. Hence the study of the performance of hardware implementation of cryptographic algorithms, especially using programmable logic devices such as Field Programmable Gate Arrays (FPGAs), has become a timely and important field of research.

1.2 Objective of the Thesis

The objective of the thesis is the performance analysis of the hardware implementation of the authentication algorithms which are widely used in Internet protocol security. The hardware utilization and timing analysis with respect to a high-end field programmable gate array device are studied using several possible optimization

techniques. The potential throughputs of the implementations are analyzed in the context of traffic characteristics of the Internet. This is done using four traffic models which have different degrees of accuracy.

1.3 Thesis Outline

The thesis consists of 7 chapters. Chapter 2 gives the background of the research and the literature review of the previous research, which are related to this study. Chapter 3 discusses the design environment and implementation choices, which are used for the research. The implementation details and results are discussed in Chapter 4. In Chapter 5 the Internet traffic modeling that was used to analyze HMAC algorithm is discussed. The new approach in message authentication is discussed in Chapter 6. Finally in Chapter 7, conclusions and future work are presented.

Chapter 2

Background of Study and Previous Research

In this chapter the background of the research and some of the key areas of application are discussed. Hash algorithms are widely used in Internet security to provide message authentication. In fact the Internet has become one of the main areas of application of cryptographic hash algorithms. Hence at the beginning of the chapter, the protocol used in Internet security is briefly described. Various studies have been carried out in the areas of cryptographic hash functions, message authentication codes and hardware implementations of cryptographic algorithms. Some of these studies are discussed in the subsequent chapters.

2.1 Internet Protocol Security (IPSEC)

IPSEC [6] is one of the key technologies for providing security as a foundation network service [12]. It is the security standard defined by Internet Engineering Task Force (IETF) for IP network layer security. According to [6], IPSEC provides security services at the IP layer by enabling a system to select required security protocols, determine the algorithm(s) to use for the service(s), and put in place a cryptographic technique to provide the requested services. The key services used to protect against the threats are confidentiality, integrity and authentication. IPSEC allows for end-to-end

encryption and authentication making TCP/IP communications secure for use in both public and private networks. The IP layer of the TCP/IP protocol architecture has been chosen as a place to operate IPSEC.

The security services offered by IPSEC include access control, connectionless integrity, data origin authentication, protection against replays, confidentiality and limited flow confidentiality. These services are provided at the IP layer offering protection for IP and /or upper layer protocols such as TCP, UDP, ICMP and so on. These objectives are achieved through the use of two traffic security protocols - the Authentication Header (AH) [13] and the Encapsulating Security Protocol (ESP) [6] - and through the use of cryptographic key management procedures and protocols. AH provides connectionless integrity, data authentication and anti-replay services. ESP provides confidentiality, limited traffic flow confidentiality and/or connectionless integrity. It optionally provides data authentication and anti-replay services as well [6]. Both AH and ESP provide access control based on the distribution of cryptographic keys and management of traffic flows relative to these security protocols. Both may be applied alone or in combination with each other to provide a desired set of services in Internet standards, IPv4 [14] and IPv6 [15].

2.1.1 Security Associations (SA)

This is a key concept fundamental to IPSEC. An SA is a one-way relationship between a sender and a receiver that affords security services to the traffic carried [16]. This is uniquely identified by a triple consisting of a Security Parameter Index (SPI), the

IP destination address and the security protocol (AH/ESP) identifier. Each IPSEC connection can provide encryption and integrity/authentication, or both. When the security service is determined, the two parties must determine which algorithms to use (e.g. DES or IDEA for encryption; MD5 or SHA-1 for authentication) [12]. Then they must share session keys. SAs are used to manage this information. To ensure interoperability and for providing management capability, some external aspects of IPSEC processing are standardized. Hence a nominal model has been described in terms of two databases: the Security Policy Database (SPD) and Security Association Database (SAD) [6]. The former specifies the policies that determine the disposition of all IP traffic inbound or outbound from a host or security gateway IP implementation. The latter database contains parameters that are associated with each security association.

2.1.2 IPSEC Packets

IPSEC defines a new set of headers to be added to IP datagrams: IP Authentication Header (AH) [13] and IP Encapsulating Security Payload (ESP) [17]. These new headers are placed after the IP header and before the layer 4 protocol (TCP or UDP). These two can be used in two modes: transport and tunnel modes. In transport mode the protocols provide protection mainly for upper layer protocols. Hence the protection extends to the IP payload. In tunnel mode the protocols are applied to tunneled IP packets, which become the payload in a new IP packet.

Authentication Header (AH)

The AH format is given in Figure 1 [13].

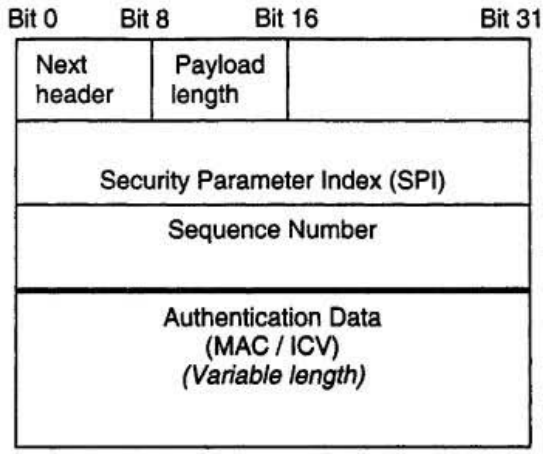


Figure 1. IPSEC Authentication Header Format

This header when added to an IP datagram ensures connectionless integrity and authenticity of the data and optionally protection against replays. This enables an end system or network device to authenticate the user or application and filter traffic accordingly. It prevents the address spoofing attacks as well. In this format the next header identifies the type of the next payload after authentication header. Payload length specifies the length of AH in 32-bit words minus 2. SPI is used to identify the SA for the datagram. Sequence number can be used as an anti replay service. Authentication data is a variable length field that contains the MAC or Integrity Check Value (ICV) for this packet. This field must be a multiple of 32-bits in length. The ICV or MAC value is computed as a function of the IP datagram and the secret authentication keying material, which is part of the SA. Only the sender and receiver know the secret keying material. If the authentication value is valid the data has come from the other party of the SA [6].

Encapsulating Security Payload (ESP)

ESP provides confidentiality and integrity services to IP datagrams. It may also provide limited traffic flow confidentiality, data origin authentication, connectionless integrity and anti replay-service for IP datagrams depending upon the implementation and header use mode (tunnel or transport). Limited traffic flow confidentiality requires selection of tunnel mode, and the encryption occurs only between an external host and the security gateway or between two security gateways. This relieves the hosts on the internal network of the processing burden of encryption and simplifies the key distribution task by reducing the number of keys. Hence it thwarts traffic analysis based on ultimate destination. The set of services provided depends on options selected. Confidentiality may be selected independent of all other services but the use of confidentiality without integrity/authentication may be subject to certain forms of active attacks. Data origin authentication and connectionless integrity are joint services and are offered as optional services. The anti replay-service may be selected only if data origin authentication is selected. The ESP header is inserted after the IP header and before the upper layer protocol header (transport mode) or before an encapsulated IP header (tunnel mode). The ESP header format is given in Figure 2. Most of the fields have similar purposes as mentioned under AH. The payload data is a variable field, which is the transport level segment or IP packet protected by encryption. The padding field is used to expand the plaintext, to conceal the actual length, or for any other alignments required by the ESP format. The Authentication Data is a variable length field that gives the MAC computed over the ESP packet minus the authentication data field.

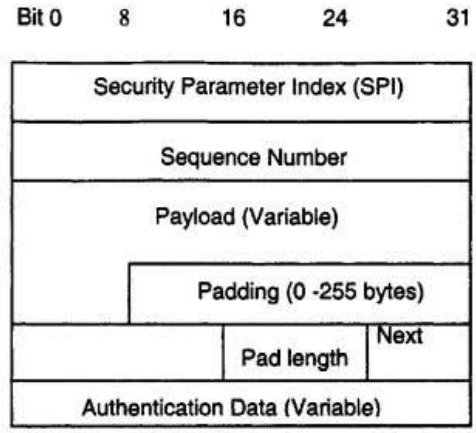


Figure 2. IPSEC ESP header format

2.2 Hash Functions

Some important algorithms and techniques resulting from various studies related to hash functions and message authentication codes are discussed in the following sections.

2.2.1 Properties of Hash Functions

A hash function is a mathematical function that maps values from a large (or very large) domain into a smaller range, and that reduces a potentially long message into a "message digest" or "hash value". It provides a fast method of storing information in and retrieving from large databases. Hash functions are used in implementing associative memories and error correction as well. With the advent of public key cryptography and digital signature schemes, cryptographic hash functions have gained more attention in

their role of providing authenticity for a message. A "good" hash function is one that results in a set of values that are evenly (and randomly) distributed over the range [18]. In order to avoid the possible attacks, a hash function used for cryptographic purposes should have several properties [19]:

- Weakly collision-free: Let M be a message. A hash function H is weakly collision-free for M if it is computationally infeasible to find a message $M' \neq M$ such that $H(M') = H(M)$.
- Strongly collision free: A hash function H is strongly collision-free if it is computationally infeasible to find messages M and M' such that $M' \neq M$ and $H(M') = H(M)$.
- One-way property: A hash function is one-way if, given a message digest Z, it is computationally infeasible to find a message M such that $H(M) = Z$.
- The input can be of any length and output has a fixed length.
- The hash function H is relatively easy to compute for any given M.

2.2.2 Digital Signatures

We now briefly describe one of the principle cryptographic applications of hash functions. Sometimes it is required to verify the origin of a document, the identity of the sender, the time and date a document was sent and/or signed, the identity of a computer or user, and so on. A *digital signature* is a cryptographic means through which many of these may be verified. The digital signature can be computed using the Digital Signature Algorithm (DSA) [20] and a set of parameters such that the identity of the signatory and

integrity of the data can be verified. The DSA provides the capability to generate and verify signatures. Signature generation makes use of a private key to generate a digital signature. Signature verification makes use of a public key, which is related to, but is not the same as, the private key. Each user possesses a private and public key pair. In general public keys are assumed to be known to the public. Private keys are never shared. Anyone can verify the signature of a user by employing that user's public key. Signature generation can be performed only by the possessor of the user's private key. A cryptographic hash function is used in the signature generation process to obtain a compressed version of data, called a message digest (Figure 3).

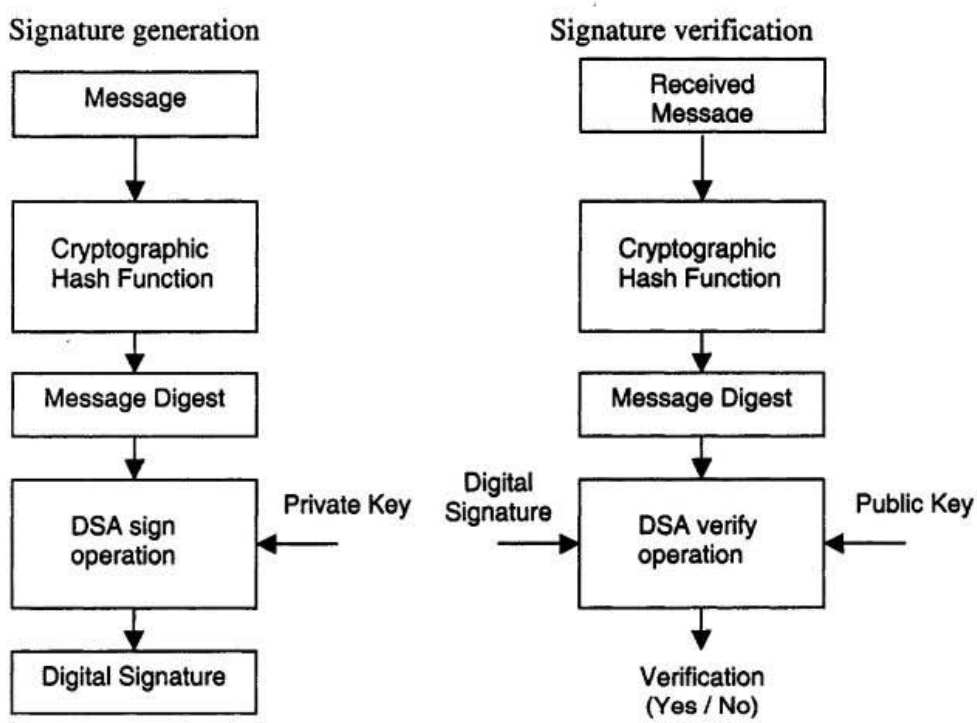


Figure 3. Cryptographic hash functions in digital signature scheme

The message digest is then input to the DSA to generate the digital signature. The digital signature is sent to the intended verifier along with the signed data (often called the message). The verifier of the message and signature verifies the signature by using the sender's public key. The same hash function must also be used in the verification process [21].

2.2.3 General Approaches to Hash Function Construction

Many researchers have discovered a number of techniques to develop hash functions. We now describe some of the approaches of constructing hash functions.

Hash function based on block ciphers

This is an effort to build hash functions on the existing block ciphers rather than constructing them from scratch. The encryption of plaintext X with key K will be denoted with $E(K, X)$. The size of the plaintext and ciphertext in bits is denoted with n and the size of the key size in bits is denoted with k. The argument of the iterated hash function is divided into t blocks X_1 through X_t. If the total length is not a multiple of n, the argument has to be padded with some accepted padding rule. If the round function is denoted by f, the hash function H can be described as follows:

$$H_i = f(X_i, H_{i-1}) \qquad i = 1,2,\ldots t.$$

where, H_0 is Initial Value (IV), specified with the scheme and H_t represents the hash code.

The general construction for the round function of the hash functions is shown in Figure 4. For simplicity it is assumed that $k = n$. The block cipher has two inputs: the key K and the plaintext P and the output C. The inputs P and K can be selected from one of

the four values: X_i, H_{i-1}, $X_i \oplus H_{i-1}$, and a constant value V. It is also possible to modify with feed forwarding (FF) the output C by XOR of one of these values. Preneel shows that these possibilities yield $4^3 = 64$ different schemes [22]. He also shows that, only 12 of these schemes are secure.

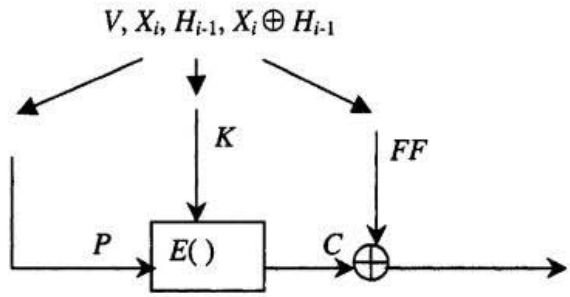


Figure 4. General model for round function of block cipher based hash function

Hash functions based on modular arithmetic

Two important hard problems in number theory, factorization and the discrete logarithm, are used to build these hash functions. These can have variable digest length depending on the size of the modulus [23]. The purpose of deploying modular arithmetic is to save on implementation cost by using existing cryptographic systems such as RSA public key cryptosystem [24]. Due to their various weaknesses, these kinds of hash functions are not popular in today's cryptographic applications.

Dedicated Hash Functions

While many hash functions are based on existing security mechanisms such as block ciphers or modular schemes, performance critical applications require the use of

functions designed for explicit use of hashing. These are called dedicated or customized hash functions [25]. Dedicated hash functions tend to be fast, achieving a considerable advantage over algorithms that are based on other techniques. These hash functions are not provably secure, as they are not based on a hard problem such as factorization. But most of them are computationally secure. Message Digest (MD)-family hash functions fall into this category, which were proposed by RSA Data Security Inc. These are iterative hash functions based on a compression function with fixed size input. The compression function consists of operations such as modular 2^{32} addition, rotation and permutation, which can be easily performed either in software or hardware. MD4 [26] is an early example of a popular hash function with such a dedicated design. Although MD4 is no longer considered secure for most cryptographic applications, most new dedicated hash functions make use of the same design principles as MD4 in strengthened versions. Their strength varies depending on the techniques, or combinations of techniques, employed in their design. Some of the popular dedicated hash functions in current use include MD5, SHA-1, RIPEMD-160 and HAVAL.

Other Approaches

Hash functions based on the Knapsack problem and Cellular Automata are some other approaches. Since hash functions never have to be inverted, completely random generated knapsacks can be used for their construction. Two examples of the hash functions based on the Knapsack problem are hash functions based on additive knapsacks and multiplicative knapsacks. [27] But it has been proven that these can be broken [28].

Wolfram has suggested a random sequence generator using cellular automata [29]. Using this pseudo random generator Daeman et al. suggested a hash function called "cellhash" [30] in which the hash result of a message M of length n is computed in two phases. In the first phase the message is extended with minimum number of zeros so that its length in bits is at least 248 or congruent to 24 mod 23. Let the resulting message be M_0, $M_1 \ldots M_{N-1}$ each of 32-bit words. Then in the second phase the hash function $F_c(H, A)$ is applied. The hash function $F_c(H, A)$ is a function with argument H, a bit string of length 257 and A, a bit string of length 256. It returns a bit string of length 257. Initial value (IV) is the all-zero bit string of length 257. The computation involves determining values for H^j:

$$H^j = F_c(H^{j-1}, M_{j-1}\, M_{jmodN} \ldots M_{j+6\ modN}), j = 1 \ldots N$$

where $H^0 = IV$ and H^N is the 257-bit hash result.

This is a hash function suitable for hardware implementations. The core is made up of two cellular automata operations and permutations. In this algorithm the diffusion and confusion properties have been obtained by linear cellular automaton and non-linear automaton respectively.

$F_c(H, A)$ has five step-transformation of H. Let $h_0, h_1 \ldots\ldots h_{256}$ denote the bits of H and $a_0, a_1 \ldots\ldots a_{255}$ represent the bits of A.

Step1: $h_i = h_i \oplus (h_{i+1} \vee \overline{h_{i+2}}) \qquad 0 \le i < 257$

Step2: $h_0 = \overline{h_0} \qquad 0 \le i < 257$

Step 3: $h_i = h_{i-3} \oplus h_i \oplus h_{i+3} \qquad 0 \le i < 257$

Step4: $h_i = h_i \oplus a_{i-1}$ $\quad 0 \le i < 257$

Step5: $h_i = h_{10^*i}$ $\quad 0 \le i < 257$

where $\oplus$, $\vee$ and $\bar{}$ represent XOR, OR and NOT operations respectively.

2.2.4 Message Digest 5 (MD5)

MD5 [4] is a message digest algorithm developed by Ron Rivest at MIT. It is basically a secure version of his previous algorithm, MD4 which is a little faster than MD5. This has been the most widely used secure hash algorithm particularly in Internet-standard message authentication. The algorithm takes as input a message of arbitrary length and produces as output a 128-bit message digest of the input. This is mainly intended for digital signature applications where a large file must be compressed in a secure manner before being encrypted with a private (secret) key under a public key cryptosystem.

Assume we have an arbitrarily large message as input and that we wish to find its message digest. The processing involves the following steps.

(1) Padding

The message is padded to ensure that its length in bits plus 64 is divisible by 512, that is, its length is congruent to 448 modulo 512. Padding is always performed even if the length of the message is already congruent to 448 modulo 512. Padding consists of a single 1-bit followed by the necessary number of 0-bits.

(2) Appending length

A 64-bit binary representation of the original length of the message is concatenated to the result of step (1). (Least significant byte first). The expanded message at this level will exactly be a multiple of 512-bits. Let the expanded message be represented as a sequence of L 512-bit blocks Y_0, $Y_1,..,Y_q,..,Y_{L-1}$ as shown in Figure 5 [16]. Note that in the figure, IV and CV represent the initial value and chaining variable, respectively.

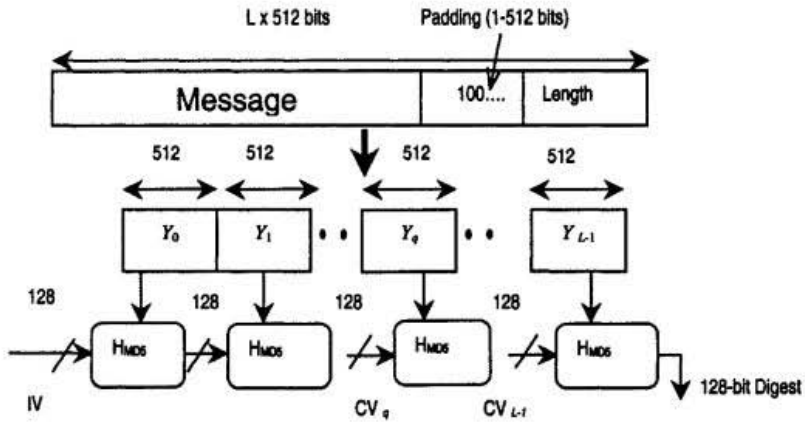


Figure 5. Generation of message digest

(3) Initialize the MD buffer

The variables IV and CV are both represented by a four–word buffer (ABCD) used to compute the message digest. Here each of A, B, C and D is a 32-bit register and they are initialized as IV to the following values in hexadecimal. Low-order bytes are put 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

(4) Process message in 16-word blocks

This is the heart of the algorithm, which includes four "rounds" of processing. It is represented by H_{MD5} in Figure 5 and its logic is given in Figure 6. The four rounds have similar structure but each uses different auxiliary functions *F, G, H* and *I*.

$$F(X,Y,Z) = (X \wedge Y) \vee (\overline{X} \wedge Y)$$
$$G(X,Y,Z) = (X \wedge Z) \vee (Y \wedge \overline{Z})$$
$$H(X,Y,Z) = X \oplus Y \oplus Z$$
$$I(X,Y,Z) = Y \oplus (X \vee \overline{Z})$$

where, $\vee$, $\wedge$, $\oplus$ and $\bar{}$ represent the logical OR, AND, XOR and NOT operations, respectively. Each round consists of 16 steps and each 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 2^{32} times abs(sin(i)), where i is in radians. The value i represents the step number. Each round also takes as input the current 512-bit block (Y_q) and the 128-bit chaining variable (CV_q). An array X of 32-bit words holds the current 512-bit Y_q. For the first round the words are used in their original order. The following permutations of the words are defined for rounds 2 through 4:

$$\rho_1(i) = i$$
$$\rho_2(i) = (1+ 5i) \bmod 16$$
$$\rho_3(i) = (5+ 3i) \bmod 16$$
$$\rho_4(i) = 7i \bmod 16$$

In Figure 6, $X[\rho_1(i)]$, $X[\rho_2(i)]$, $X[\rho_3(i)]$ and $X[\rho_4(i)]$ represent 16 words of X, permuted according to $\rho_1(i)$, $\rho_2(i)$, $\rho_3(i)$ and $\rho_4(i)$ respectively.

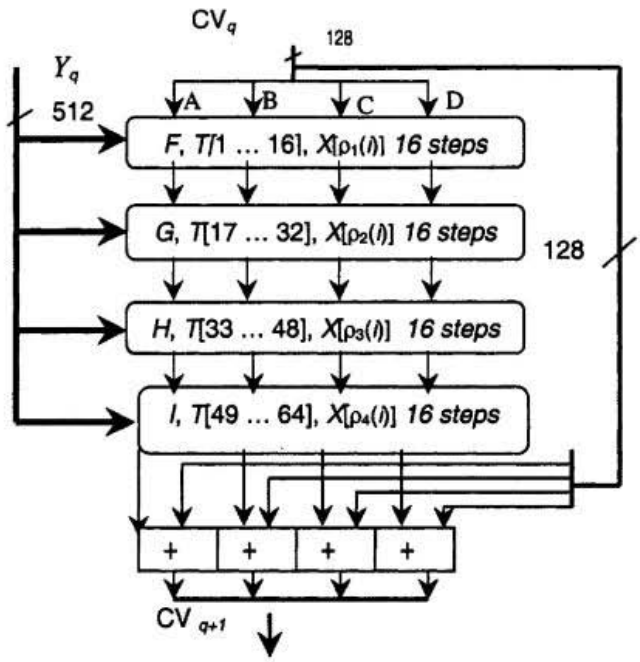


Figure 6. Compression function H_{MD5}

The output of the fourth round is added to the input of the first round (CV_q) to produce CV_{q+1}.

(5) Output

After all L 512-bit blocks have been processed, the output from the L^{th} stage is the 128-bit message digest.

Figure 7 shows the operations involved in a single step [16]. The additions are performed as modulo 2^{32} operations. Four different circular shift amounts (s) are used

each round and these are different from round to round. Here $X[k]$ represents the corresponding word for the step according to the permutation rule mentioned earlier.

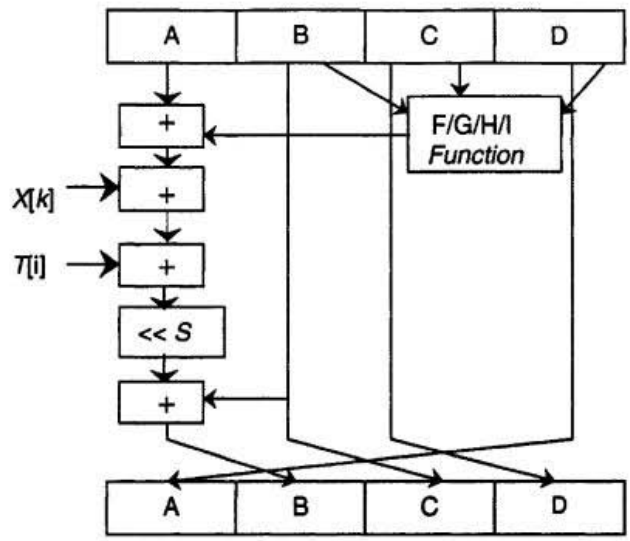


Figure 7. Operations in a single step of MD5

The operation of a step can be represented mathematically as follows:

$$A \leftarrow D$$
$$B \leftarrow B + ((A + Func(B, C, D) + X[k] + T[i]) << S)$$
$$C \leftarrow B$$
$$D \leftarrow C$$

2.2.5 Secure Hash Algorithm-1 (SHA-1)

SHA-1 is the algorithm specified in the Secure Hash Standard [8], which was developed by NIST. When a message of any length $< 2^{64}$ bits is input, SHA-1 produces a 160-bit output as a message digest. The overall processing of a message follows the MD5 structure given in Figure 5 with H_{MD5} and the hash / chaining variable lengths replaced with $H_{SHA\text{-}1}$ and 160 bits, respectively. The processing consists of following five steps.

(1) Padding

The message is padded to ensure that its length in bits plus 64 is divisible by 512. That is, its length is congruent to 448 modulo 512. Padding is always performed even if the length of the message is already congruent to 448 modulo 512. Padding consists of a single 1-bit followed by the necessary number of 0-bits.

(2) Appending length

A 64-bit binary representation of the original length of the message is concatenated to the result of step (1). (Most significant byte first). The expanded message at this level will exactly be a multiple of 512-bits. Let the expanded message be represented as a sequence of L 512-bit blocks $Y_0, Y_1, \ldots, Y_q, \ldots, Y_{L-1}$ as shown in Figure 5 [16].

(3) Initialize

The variables IV and CV are represented by a five–word buffer (ABCDE) used to compute the message digest. Here each of A, B, C, D and E is a 32-bit register and they are initialized as IV to the following values in hexadecimal. These values are stored in big-endian format, which has the most significant byte of a word in the low-address byte position.

A: 67452301

B: EFCDAB89

C: 98BADCFE

D: 10325476

E: C3D2E1F0

(4) Process message in 16-word blocks

As in MD5, this is the heart of the algorithm, which includes four "rounds" of processing and its logic is given in Figure 8. Each round has 20 steps.

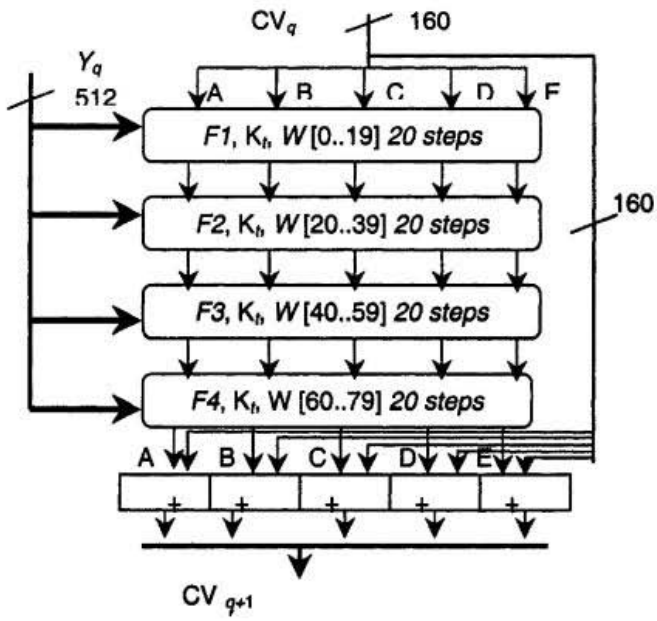


Figure 8. Compression Function H_{SHA-1}

The four primitive functions are,

$$F1(X,Y,Z) = (X \wedge Y) \vee (\overline{X} \wedge Y)$$
$$F2(X,Y,Z) = X \oplus Y \oplus Z$$
$$F3(X,Y,Z) = (Y \wedge Y) \vee (X \wedge Z) \vee (Y \wedge Z)$$
$$F4(X,Y,Z) = X \oplus Y \oplus Z$$

The four rounds have similar structure but each uses different auxiliary functions *F1* (steps $0 \le t \le 19$), *F2* (steps $20 \le t \le 39$), *F3* (steps $40 \le t \le 59$), and *F4* (steps $60 \le t \le$

79). Each round utilizes an additive constant K_t, where $0 \leq t \leq 79$ indicates the step. Unlike MD5, SHA-1 uses only four distinct constants. The constants in hexadecimal are as follows.

Step	K_t in Hexadecimal
$0 \leq t \leq 19$	K_t=5A827999
$20 \leq t \leq 39$	K_t=6ED9EBA1
$40 \leq t \leq 59$	K_t=8F1BBCDC
$60 \leq t \leq 79$	K_t=CA62C1D6

Each round also takes as input the current 512-bit block (Y_q) and the 160-bit chaining variable (CV_q). Then CV_q is updated through the four rounds and the output of the fourth round (80^{th} step) is added to the input to the first round to produce CV_{q+1}. This addition is done independently for each of the five words in the buffer with each of the corresponding words in CV_q using modulo 2^{32}.

(4) Output:

When all the blocks have been processed, the 160-bit output will be the message digest. The generic step of the compression function is shown in Figure 9.

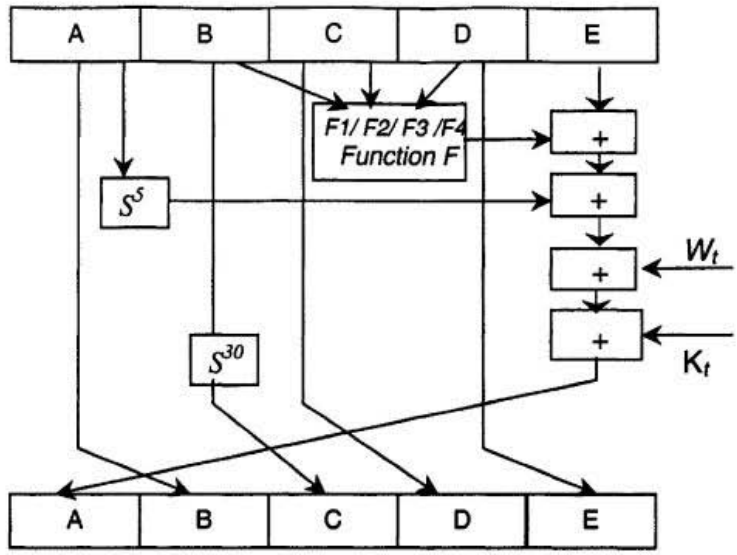


Figure 9. Operations in a single step of SHA-1

The operations in a step can be represented in mathematical form as follows.

$$A \leftarrow E + F(B,C,D) + S^5(A) + W_t + K_t$$
$$B \leftarrow A$$
$$C \leftarrow S^{30}(B)$$
$$D \leftarrow C$$
$$E \leftarrow D$$

Here, A, B, C, D and E are five words of the input buffer. S^n denotes the circular left shift of the 32-bit argument by n bits. K_t is the additive constant for step t. W_t represents 32-bit word derived from the current 512-bit input block. All additions (+) are modulo 2^{32} additions.

2.3 Message Authentication Codes (MACs)

Message authentication is the procedure to verify that received messages come from the alleged source and have not been altered. A MAC is an authentication tag (checksum) derived by applying an authentication scheme, together with a secret key, to a message so that the recipient can verify the message authenticity. There are several types of construction available to produce MACs. These include

(a) Block cipher based MACs

(b) Keyed hash function based MACs

(c) Universal hash function based MACs.

In our work we have focused on the important technique of combining a key with an unkeyed hash function, referred to as HMAC which is of type (b). Apart from these methods there are other techniques as well. Unconditionally secure MACs [19] based on encryption with a one-time pad and stream cipher based MACs [31] are other examples. A MAC is said to be secure if it can resist existential forgery under an adaptive chosen-message attack [32].

2.3.1 Block Cipher Based MACs

Cipher Block Chaining MACs (CBC-MACs)

This is the most widely used MAC, first used in the mid 1970s. A block cipher is used in Cipher Block Chaining mode as shown in Figure 10. Let a message space for M

be binary strings whose lengths are a positive multiple of l. Hence the message M can be broken into blocks such that

$$M = M_1, M_2, .., M_m \text{ with } |M_i| = l.$$

Then each block is passed through the encryption E with key K and the result is XORed with the next block. E_K represents the encryption using a key K. Cipher block chaining is given by

$$\text{CBC}_{E_K}(M) = \sum_{i=1}^{m} E_K[M_i \oplus C_{i-1}] \qquad \text{For } i = 1... \ m \text{ and } C_0 = 0^l$$

where, 0^l indicates l bit vector of all zeros.

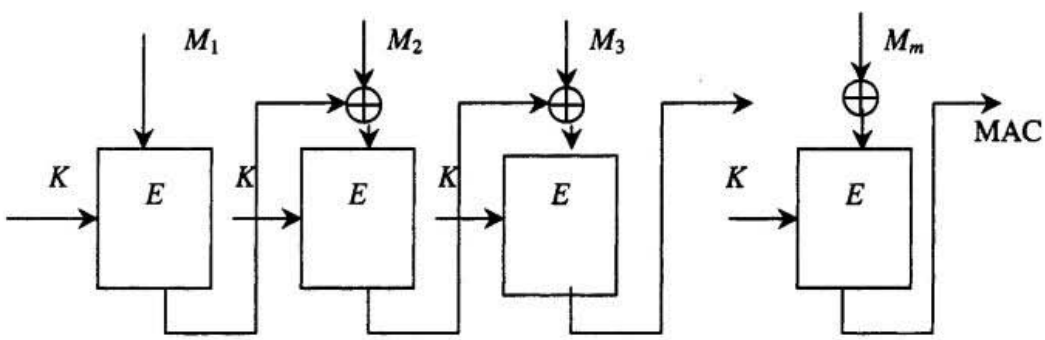


Figure 10. CBC-MAC

The standards ANSI X9.9 [33] and FIPS 113 [34] describe CBC-MAC. Although it was proven secure for fixed length messages by Bellare et al. [35], CBC-MAC does not directly give a method to authenticate messages of variable lengths. If the length of strings varies from a multiple of block length, the CBC-MAC becomes more vulnerable.

Bellare suggested three methods: input-length key separation, length prepending and encrypting last block to overcome that vulnerability [32].

XOR MACs

This is an approach for authenticating a message using a Finite Pseudo Random Function (PRF). This PRF can be instantiated with a block cipher or a compression function of a cryptographic hash function. The advantages of this are parallelizability, incrementability and provable security. There are three main steps in this algorithm [36].

1. Encode message M as a collection of blocks.
2. Apply the finite PRF to each block creating collection of PRF images.
3. XOR the set of PRF images together.

XOR MAC has two schemes namely, randomized and counter based. In the randomized scheme, it is assured that there is a PRF F with input size l, output size L and key length k bits. A parameter b is fixed where $b < l$, which is the block size. Assume the message to be authenticated M has length at most $b \times 2^{l-b-1}$. By standard padding arguments we may assume that the message is a multiple of b. Usually b is taken as $l/2$. So

$$M = M_1, M_2, \ldots M_n$$

where $|M_i| = b$. Let $<i>$ denote the binary representation of the l-b-1 bit encoding of the integer i. It is the binary representation of the block index $i \in \{1,2,\ldots n\}$.

Assume $K \in \{0, 1\}^k$ is the shared key of length k. To compute the XOR MAC of a message the sender chooses a random l-1 bit string r (known as *seed*) and then computes the tag(r, t) using

$$t = F_K(0 \| r) \oplus F_K(1 \| <1> \| M_1) \oplus F_K(1 \| <2> \| M_2) \oplus \ldots F_K(1 \| <n> \| M_n).$$

Here || denotes concatenation and F_K represents a finite pseudo random function such as a block cipher (e.g. DES) with a key K. Set the MAC of M to the pair (r, t). Thus the sender transmits (M, r, t). The receiver verifies the message by computing t' from M and r (and K). This is a parallelizable structure and hence suitable for long messages and/or expensive PRFs. The other feature is the MAC is incremental with respect to substitutions. When it is computing the tag for a related message (e.g. only one block is different) only a constant amount of computation is required [36].

In counter based XOR scheme, a sender maintains an n-bit counter C which is initially 0 and incremented for each message. Authentication has to follow these steps.

- increment the counter C by 1
- set $t = F_k(0 \| c) \oplus F_k(1 \| <1> \| M_1) \oplus F_k(1 \| <2> \| M_2) \oplus \ldots F_k(1 \| <n> \| M_n)$
- set the MAC of M to (c, t)

The sender transmits (M, c, t). The receiver calculates t' using the received (M', c', t') and accepts the message if $t = t'$.

2.3.2 Keyed Hash Function Based MACs

A function $H()$ that maps a fixed length key K and an arbitrary length message M to a n-bit message digest MD is a keyed hash function, if it satisfies the following properties [23].

- The description of $H()$ is publicly known.
- Given K and M, it is easy to compute $H(K, M)$.

- Without knowledge of K, it is hard both to find M when $H(K, M)$ is given, and to find two distinct messages M and M' such that $H(K, M) = H(K, M')$.
- Given (possibly many) pairs of $\{M_i, MD_i\}$ with $MD_i = H(K, M_i)$, it is hard to find the secret key K.
- Without knowledge of K, it is hard to determine $H(K, M)$ for any message M, even when a large set of pairs $\{M_i, H(K, M_i)\}$, where M_i s are selected by the opponent ($M \neq M_i, \forall M_i$), is given.

Keyed hash functions can be constructed from existing hash functions. Hence some security requirements of the designed keyed hash function rely on the security of the underlying hash function. Basically three methods have been adopted in constructing keyed hash functions [23]. "Hash then encrypt" is the simplest keyed hash function which can be defined as

$$H(K, M) = E(K, H'(M))$$

where K is the secret key, M is the message, H' is the one-way hash function and E is the encryption algorithm.

The second method is the nested hash function. There are many schemes of constructing keyed hash functions using this method. In these schemes the hash function introduces a key by setting it as the initial vector. Hash functions are nested with different initial vectors. Suppose H' is a one way hash function. The round function f can be defined as

$$f(X_i, M_i) = H'(H'(X_i) \| M_i).$$

where, M_i is the i^{th} message block, X_i is the chaining variable (output of the last round), "||" represents concatenation and $X_1 = IV'$, which is set to the key K. This is shown in Figure 11. The keyed hash function would be

$$H(K, M) = f(f(...f(f(K, M_1), M_2),..., M_{n-1}), M_n).$$

It has been found that this scheme does not satisfy the fourth property required for a keyed hash function. This also suffers from padding attack, which is explained in section 2.4.3. Hence a modified keyed hash algorithm was developed which introduces a new "salt value" S. The construction is given Figure 12. The H'' can be built using the round function explained earlier. In these schemes the several hash operations increase the overhead and decrease the hash throughput.

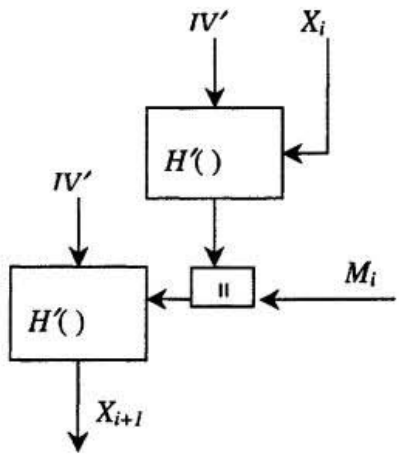


Figure 11. Round function using nested hash functions

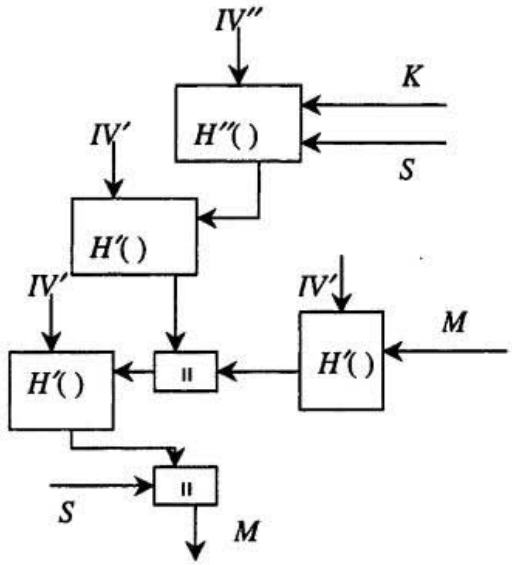


Figure 12. Modified keyed hashed function with nested hash functions.

Other methods using a key as part of a message or initial vector have been proposed. In doing so the speed of the constructed hash function remains almost as fast as the hash function. Many schemes have been developed according to this method.

Tsudik [37] suggested three methods: secret prefix, secret suffix and envelop method. These methods are only based on one-way hash functions and use keys to make them secure. If H' is a collision free hash function, K (= $K_1 \| K_2$) is a secret key, then the keyed hash function $H(K, M)$ is defined as follows.

Secret prefix method:

$$H(K, M) = H'(IV, (K \| M)).$$

This method is not susceptible to the birthday attack, which is explained in 2.4.1. But since the attacker knows the underlying hash algorithm and the integrity value is not protected, there can be a padding attack (see section 2.4.3). A counter measure for this attack is to include the message length in the secret prefix calculation, as part of the first block. Some protocols such as Simple Network Management Protocol (SNMP)[38] utilize this method.

Secret suffix method:

$$H(K, M) = H'(IV, (M \| K)).$$

This is suitable for high bandwidth and low delay requirements such as packetized voice and video applications. One drawback of the secret suffix method is the susceptibility to the birthday attack (see 2.4.1).

Envelope method:

$$H(K, M) = H'(IV, (K_1 \| M \| K_2)).$$

The message is enveloped with a secret prefix and a secret suffix before its digest is computed. This is claimed to be more secure than the previous methods and is resistant to both padding and birthday attacks.

In all these three methods the key size would be either the block size of the message or twice that. Hence the processing speed is reduced by extra block processing time. The security solely depends on the underlying hash function. A detailed analysis of these methods can be seen in [39]. In that Preneel et al. also have suggested a heuristic construction, MD_x-MAC which is free from some of the previous weaknesses. This trend was continued further by Bakhtiari et al. as in [40]. They introduced six improved methods using small keys (128 bits). In these schemes $H'(\ )$ is a collision free hash function, $K=(K_1 \parallel K_2)$ is the secret key, $\overline{\oplus}$ and $\underline{\oplus}$ are special XOR operations between two entities (X and Y) of different lengths as described below:

$X \overline{\oplus} Y$ - The shorter one (between X and Y) is padded by zeros from the right hand side to make its length same as the other and then they are XORed.

$X \underline{\oplus} Y$ - The shorter one is padded by zeros from the left hand side to make its length same as the other and then they are XORed.

The methods are summarized below.

$$(1) H(K,M) = H'(IV,(K \overline{\oplus} M))$$
$$(2) H(K,M) = H'(IV,M)$$
$$(3) H(K,M) = H'(IV,(M \underline{\oplus} K))$$
$$(4) H(K,M) = H'(IV,(K \overline{\oplus} M \underline{\oplus} K))$$
$$(5) H(K,M) = H'(IV,(M \underline{\oplus} K)) \quad \text{where } IV = K$$
$$(6) H(K,M) = H'(IV,(M \underline{\oplus} K_2)) \quad \text{where } IV = K_1$$

The concatenation of the key to the message in earlier methods has been eliminated by using XOR operation. Since the key does not increase the length of the input, processing speed is not affected. But some of these methods have some weaknesses. For example (1) and (2) methods suffer from padding attack. The authors claimed that the method (5) would be the best as it is safe enough against the possible attacks and at the same time is efficient.

In 1996 M. Bellare et al.[3] presented two constructions, which could be formally analyzed without resorting to unrealistic assumptions such as "idealness" of the underlying hash functions. Their approach of keying the hash function is to substitute the secret key for the functions fixed initial value *IV*. The fixed and known *IV* of the original function is replaced by a random and secret value *K* known only to the communicating parties. The two constructions proposed by them were nested MAC (NMAC) and the hash-based MAC (HMAC). In these schemes the iterated construction methodology for construction of collision–resistant hash functions [27], is utilized. The input is hashed by iterating the compression function. If the input is *M*, then

$$M = M_1, M_2, \ldots M_n$$

where $|M_i| = l$. The value of iterated function *F* on *M* is h_n, where

$$h_0 = IV$$

$$h_i = f(h_{i-1}, M_i).$$

Their approach is to substitute the secret key to the functions fixed *IV*. Let the keyed compression function and its iterated function be defined as f_K and F_K, respectively. If K= (K_1, K_2) where K_1 and K_2 are the keys to the function of length l each:

$$\text{NMAC}(M) = f_K(M) = (F_{K1}(F_{K2}(M))$$

Here the outer function acts on the output of the iterated function and hence involves only one iteration of the compression function. That is, the outer function is the compression function acting on the $F_{K2}(M)$ padded to a full block size. So, even though NMAC construction is simple and efficient, it requires direct access to the code for the compression function rather than the overall hash function. HMAC was suggested to avoid that requirement [3].

In HMAC the two keys are derived pseudorandomly from a single key, which is an advantage at the level of key management. Let F be the key-less hash function with initial value *IV*. The arbitrary length message is processed with random string K of length l as follows

$$\text{HMAC}(M) = F(K' \oplus \text{opad}, F(K' \oplus \text{ipad}, M))$$

where K' is the padded K with required '0's to a one b-bit block size of the iterated hash function. Here, ipad is the byte 0X 36 and opad is the byte 0X 5C each repeated 64 times. These values are chosen to have high hamming distance between the pads. They are set to exploit the mixing properties attributed to the compression function underlying the hash schemes in use. These are important properties to provide computational independence

between the two derived keys. The complete implementation and performance analysis of HMAC based on SHA-1 is discussed in Section 4.4.

2.3.3 Universal Hash Function Based MACs

The idea of the universal hash function was introduced by Carter and Wegman in 1979 [41]. Two years later they introduced MACs using universal hash families [42]. According to this, the message M is hashed to a smaller size using a function from a universal hash function family, which has only a combinatorial (rather than a cryptographic) property. Then a cryptographic primitive with one-time pad and encryption is applied to the resulting smaller string to produce the MAC as in Figure 13 [43].

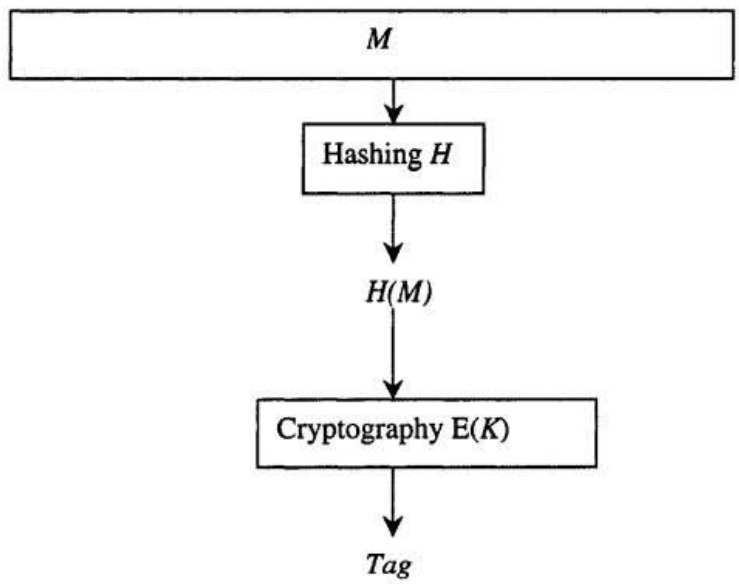


Figure 13. Carter-Wegman MACs

By definition, a finite multiset of hash functions $H=\{h: A \rightarrow B\}$ is said to be universal if for every $x, y \in A$ where $x \neq y$: $\Pr_{h \in H}[h(x) = h(y)] = 1/|B|$. Further, a finite multiset of hash functions $H=\{h: A \rightarrow B\}$ is ε-Almost Universal (ε-AU) if for all $x, y \in A$ where $x \neq y$, $\Pr_{h \in H}[h(x) = h(y)] \leq \varepsilon$.

The above approach has been suggested as a promising way for a highly secure, ultra-fast MAC. The speed of universal hashing MAC depends on the speed of hashing step and the speed of the encryption step. If hash function compresses messages well, encryption will be faster. The combinatorial property of the universal hash function family can be mathematically proven without cryptographic hardness assumptions.

Several fast-to-compute hash functions have been developed. Among them, the divisional hash is a method related to Cyclic Redundancy Check (CRC) computations, which is shown to be ε-AU for a small ε. A Linear Feedback Shift Register (LFSR) based Toeplitz hash is another method, which is based on matrix-multiplication. Both these methods allow efficient hardware implementation [43]. In 1995 Rogaway suggested bucket hashing [44], which is used in the context of Wegman-Carter authentication [42]. This has been explicitly targeted for software implementation. Halevi and Krawczyk proposed Multilinear Modular Hashing (MMH) and Nonlinear Modular Hashing (NMH) which take full advantage of microprocessor technologies such as Intel's MMX and 64-bit architecture, to achieve Gbps rates [45]. The Universal Message Authentication Code (UMAC) was proposed in 1999 [46]. This has been targeted for high throughput software implementations. Unlike traditional inherently serial MACs, UMAC is parallelizable. The

underlying hash function family is known as New Hash (NH), which is a simplification of MMH and NMH. The details of this algorithm are discussed in Chapter 6.

2.4 Attacks on Hash Functions

A successful attack on a hash function means finding a method to falsify a claimed security property of the hash function. There are two main groups of attacks: general and special attacks. The special attacks depend on the weaknesses of the underlying algorithm and the general attacks are independent of the algorithm.

2.4.1 General Attacks

This group of attacks only depends on the message digest length. The birthday attack, exhaustive key search, pseudo key attack and random attack are some examples of this group.

Birthday attack:

This attack is one of the most powerful attacks on hash functions with uniform message digest distribution and short message digest length. This is based on the "birthday paradox" which is a label for the following phenomenon: Given a random variable that is an integer with uniform distribution between 1 and N and a selection of k instances ($k \leq N$) of the random variable, it can be shown that[16]

$$P(N, k) > 1 - e^{-(k(k-1))/2N}$$

where, P(N, k) is the probability that there is at least one duplicate. According to this, the value of k required for P(N, k) > 0.5 can be shown as $1.18(N)^{1/2}$, which is approximately $(N)^{1/2}$. The birthday paradox gets its name from a special case with N =365 for which the value of k is ≈ 19. This means that the minimum number of people required for at least two people have the same birthday with the probability about 0.5 is 19. Suppose the message digest of m bits is produced by a hash function on the message M. According to the birthday paradox, if the hash function is applied to k random inputs, the value of k so that there is a probability about 0.5 for at least one duplicate will be $2^{m/2}$. The adversary creates a pool of many message and digest pairs. When the attacker intercepts a message digest, it is compared with all message digests in the pool. In the case of a match, the corresponding message is sent instead of the original message. If an adversary generates r_1 variations on a bogus message and r_2 variations on a genuine message, the probability of finding a bogus message and a genuine message that hash to the same result can be approximated by

$$P \approx 1 - e^{-r_1 r_2 / 2^m}$$

where $r_2 >> 1$ [47]. When $r_1 = r_2 = 2^{m/2}$, the above probability is about 0.5. To achieve security against a birthday attack, the hash value should be at least 128 bits [48].

Exhaustive key search

If the adversary has access to at least one message-digest pair, the key can be found by examining the key space elements against the message-digest pairs. As the message space does not have a one-to-one map to digest space, more than one key could

be found. However there is a possibility to determine the key if a large number of pairs is given [23].

Pseudo Attack

The adversary tries to find a pseudo key $\overline{K}$ with $H(K, M) = H(\overline{K}, M)$ where H is the keyed hash function, K is the actual key and M is the message. This is similar to finding more than one key in exhaustive key search and may allow the enemy to identify himself as a legitimate user.

Random Attack

In this attack the adversary chooses a random message (or part of a message) and expects that its message digest is equal to a genuine one. If the message digest length is r then the success probability of this attack for a hash function, which has the required random behavior, is $1/2^r$. By having at least 128 bits for the message digest this attack can be thwarted [23].

2.4.2 Special Attacks

These attacks depend on the round function or in general the hash function design. These are not successful in keyed hash functions as the key protects the hash components against outsiders. Examples of this attack are meet-in-the middle attack, correcting block attack, fixed point attack, attack on the underlying block cipher, and differential and linear cryptanalysis. The details of these attacks are given in [48] and [40].

2.4.3 High Level Attacks

These are attacks on hash functions when used in a protocol or for non-hashing purposes. Examples are the replay attack and padding attacks. Attaching time stamps to the messages can thwart the replay attack. In padding attacks the intruder tries to append (or prepend) a message to the existing one such that the legitimate parties would accept the result. If $[M, MD]$ with $MD = f(M, K)$ is sent to the receiver, the intruder tries to find M' such that $[(M \| M'), MD]$ or $[(M' \| M), \text{MD}]$ is sent instead of $[M, MD]$ [40]. Padding attacks can be thwarted by pre-pending the message length to the message or by using some fixed suffixes that do not appear within the message.

2.5 Conclusion

We discussed the background of this research with the brief description of the IPSEC, which is one of the main areas of application of hash algorithms. Next, a detailed investigation of hash functions and message authentication codes that have been developed so far was carried out. In this section, many popular cryptographic hash functions and message authentication codes were elaborated. The attacks on hash functions were described there after.

Chapter 3

Design Environment and Implementation Choices

The main objective of this research is to develop an efficient hardware design for the hash algorithms widely used in Internet security, to give maximum speed and minimum hardware utilization. In this chapter we discuss some of the main issues that have to be considered in this effort.

3.1 Hardware vs. Software Implementation.

Today, software implementations of cryptographic algorithms are more prevalent than hardware implementations. They provide more flexibility since any algorithm can be executed on a processor. They allow ease of upgrading, ease of use and portability. But there is a growing trend of many companies in security business in developing multifunctional cryptographic accelerators. The hardware implementation of cryptographic algorithms is thriving in the new century because of the growing requirement for high speed, high volume secure communications combined with physical security [53]. Hardware implementation is more attractive due to the fact that it can take advantage of bit level and instruction-level parallelism that is not accessible to general-purpose processors [54]. A software implementer is trying to efficiently express an algorithm in terms of an existing hardware device. Hence the speed of the software

implementation is restricted to the speed of the computing platform. Whereas a hardware implementer is designing a device to perform the algorithm that has far more degrees of freedom. He can explore different versions of the same design as alternatives. This is clear when exploiting the parallelism of an algorithm. In software, the available execution units of a processor are used to maximize performance. A hardware implementation can be designed to best exploit the inherent parallelism of an algorithm. As well, hardware implementation can be optimized for speed or size. In this case size translates much more directly into cost than in the case for software implementation.

Primarily hardware implementations can be targeted for two general technologies: custom, mask-layered technology and re-configurable technology. Both of these technologies have their own advantages and disadvantages. Typically these two are known as Application Specific Integrated Circuits (ASICs) and Field Programmable Devices (FPDs).

3.2 Implementation Using Custom Hardware

ASIC may be further categorized as full custom IC and semi custom IC. Full custom ICs require the development of all the mask layers at the transistor level. Since it does not use pre-compiled, pre-characterized cells, it is a time consuming process. The full custom ASIC design methodology offers high system performance since special attention can be given to critical devices and interconnections. System performance can be optimized by sensibly controlling factors such as device location, transistor sizing and interconnecting (routing) length [55]. This process takes a long time and more personnel.

Hence this is expensive to manufacture and to design. The manufacturing lead-time is about 8-weeks. Thus full custom ASICs are generally pursued only for performance critical designs or high volume products, which can regain the initial investment.

In semi-custom ASICs all of the logic cells are predesigned and some (or all) of the mask layers are customized. There are two types of semi custom ASICs: standard cell-based ASICs (CBICs) and gate array based ASICs [56]. CBICs use predesigned standard logic cells. These cells can also be used along with larger predesigned cells known as mega cells such as micro controllers. The designer can place a cell anywhere on the silicon and hence all the mask layers are unique to a particular customer. The advantages of CBICs are saving time and reducing risk by using standard cells. The main disadvantages are the cost of standard cell libraries and time needed to fabricate all layers for each new design.

In gate-array (GA) based ASICs the transistors are predefined as a base array on the silicon wafer by replicating a base cell. Only the top few layers of metal, which define the interconnect between transistors, are defined by the designer using custom masks. To distinguish these from other types of gate arrays, this type of ASIC is often known as Masked Gate Array (MGA) [56]. The designer chooses pre-designed gate array cells known as macros from a library. The base-cell layout is the same for each logic cell, and only the interconnect (inside cells and between cells) is customized. There are three types of GAs; channeled, channel-less and structured GAs. The costs for all the initial fabrication steps for an MGA are shared for each customer and this reduces the cost of an MGA compared to a full-custom or standard-cell ASIC design. These custom ASICs

provide a specific functionality for a particular design. The design must be implemented all the way from the behavioral description to the physical layout and sent for the expensive and time-consuming fabrication [56].

3.3 Field Programmable Devices (FPDs)

FPDs have a fast design turnaround. FPGAs are a complex version of FPDs. They have no customized mask layers. The core is a regular array of programmable logic cells that can be implemented combinationally or sequentially.

Basically the FPDs can be divided into three categories. Simple Programmable Logic Devices (SPLDs), Complex Programmable Logic Devices (CPLDs) and FPGAs [57].

SPLDs

All small FPDs including Programmable Logic Arrays (PLAs), Programmable Array Logic devices (PALs) and PAL-like devices come under this category. These are suitable only for small designs.

CPLDs.

This is a large capacity device based on SPLD architectures interconnected on a single chip. For commercial CPLDs the main switching methods are Erasable Programmable ROM (EPROM) and Electrically Erasable PROM (EEPROM). Both methods use floating gate transistor technology. Due to rapidly growing market, there are many CPLD products including the Altera Max5000, 7000 and 9000 series and the Xilinx XC9000 series. CPLDs provide logic capacity of about 50 typical SPLD devices. So they can have

around 9000 system gates. They are widely used for many applications including high-speed networking, power conscious portable designs and in-system programming applications.

FPGAs

FPGAs consist of an array of uncommitted circuit elements (logic cells) and interconnecting resources. The end user can configure it through programming. There are two types of logic cells: multiplexer based (e.g. Actel) and lookup-table based (Xilinx, Lucent). A basic logic cell has a fixed number of inputs and outputs and can implement a certain set of functions. There are two classes of commercial FPGAs depending on the switching technology: antifuse and Static-RAM (SRAM). Actel, Cypress, Crosspoint, Quicklogic are some devices of antifuse technology. Antifuse-based devices are programmed once and hold their programs across power cycles and are not mutable once programmed. Xilinx, Altera-Flex, Atmel's CLI family, Toshiba are some examples of SRAM switching technology. These devices have the advantage of in-circuit re-programmability, but must be programmed each time they are powered up and hence the configuration data has to be stored in an external ROM [57]. FPGAs promote short time to market, high flexibility with capability for frequent modifications of hardware and low development cost. For cryptographic application, they also have a capability to allow for time sharing of one integrated circuit. Hence they provide many advantages for vendors and users of cryptographic equipment [53]. Due to the immense flexibility of FPGAs by incorporating a large amount of routing resources into a device, the gate-to-gate delays in such devices are higher than those of ASIC devices. But it is still possible to overcome

such drawbacks by using more parallelism in the FPGA designs. Unfortunately complex designs tend to create many more logic levels in FPGAs than ASICs and become difficult to debug. As well, much of the mixed-signal functionalities available in ASICs are not common in FPGAs. But today's highly competitive market, the first product to market establishes strong market share. In this case FPGAs provide an alternative that save designers time in the final verification cycle and in the long ASIC design process.

3.4 FPGA Implementation of Cryptographic Algorithms

In general, hardware implementation can achieve superior performance compared to software implementation. FPGA implementation is a highly promising alternative for implementing cryptographic algorithms. The fine granularity of FPGAs matches extremely well the operations required by most of the cryptographic algorithms. Especially the basic operations involved in private key cryptographic algorithms such as bit-permutation, byte substitution, lookup table readings and boolean operations can be implemented in FPGAs more efficiently than in a general purpose computer. As well the inherent parallelism of the algorithms can be efficiently exploited in FPGAs but not in the serial computing of a uniprocessor environment. This was widely studied during the process of developing an Advanced Encryption Standard (AES) by the National Institute of Standards and Technology (NIST) [49] in the U.S. Apart from the rigorous security analyses, AES candidate algorithms were studied for efficiency in both hardware and software implementations. It has been seen that use of a simple cipher design with simple

operations that possesses both cryptographic and good overall cipher efficiency is desirable for FPGA implementations [50].

The AES candidate algorithms involve boolean operations, modulo 2^{32} addition and subtraction, fixed point shifting, variable rotation, modulo 2^{32} multiplication, Galois Field 2^8 multiplication and lookup tables [51]. Modern FPGAs have a structure of two-dimensional array of configurable logic units interconnected via a large routing matrix. Configurable logic units are comprised of look up tables and flip-flops. Lookup tables can be configured as either combinational logic or memory elements. Modern FPGAs have variable size RAM blocks, which can be used as memory elements or as look up tables. The most complex operations of the block ciphers are the modulo 2^{32} multiplication and the variable rotation. The substitutions or S-boxes can be implemented in either combinatorial logic or embedded RAM blocks. However due to the limited bit width, limited number of RAM blocks and the higher switching time of the RAM compared to that of a standard CLB slice elements, the latter option is not feasible [51]. When a cipher consists of larger S-boxes and more complex operations it becomes more resource intensive. The basic operations such as bit-wise XOR, modulo 2^{32} addition and subtraction and fixed value shifting are implemented from simple hardware elements and hence they are fast. The Galois field multiplication in AES candidates is implemented efficiently in hardware as it involves multiplication by a constant [51]. It utilizes fewer resources than general multiplication. At the cryptographic-round level of the AES candidate block ciphers, multiple operations can be executed concurrently. Some operation modes (e.g. Electronic Code Book mode) allow concurrent processing of

multiple blocks of data. Hence if multiple rounds are implemented, any desired speed up of throughput can be achieved compared to a single round implementation. FPGA also provides agile key-context switching. This is the ability to generate key dependent data in early rounds before the data is required. This avoids the excessive latency in context switching that appears in software implementations [52].

There are several other potential advantages of cryptographic algorithms implemented using FPGAs. Algorithm agility is one such benefit. Many security protocols such as SSL and Internet Protocol Security (IPSEC) allow multiple encryption/authentication algorithms. These are generally negotiated on a per-session basis. For instance, IPSEC allows 3DES, Blowfish, CAST, IDEA, RC4 and RC6 as algorithms with future extensions. Algorithm agility offers the capability of switching of cryptographic algorithms during operation. It is possible to upgrade a programmed device with a new algorithm, which did not exist (or was not standardized) during design stage. Swapping a standardized algorithm with a proprietary one or changing the mode of operation are possible as well.

3.5 Device Selection

When examining cryptographic hash functions for hardware implementation, various key aspects involving the selection of the target device, design development environment and the design architectures emerge. It is clear that the number of input / output (I/O) pin requirement would be a significant factor as the algorithms deal with large data streams and the keys of minimum size of 64-bits (in case of HMAC-MD5)

[58]. The output also has at least 64 bits (truncated HMAC-MD5 output). The MD5 algorithm utilizes 64 constants each of 32 bits in length. It would be reasonable for them to be stored in a ROM. As well the necessity of exploring different architectures and optimizing the design using various techniques, results in selecting a resource-rich device. It should be able to provide the large amount of hardware resources and it should be flexible so that the design can be optimized. Although the cost of high-end FPGAs is relatively high, the rapid pace of developing these devices may result in the decrease of the FPGA cost in the future market. Therefore a high-end FPGA device was selected for this design. Based on the above requirements, the Xilinx Virtex XCV1000FG680-6 was chosen as the target device.

3.5.1 Virtex Architecture

The XCV1000 virtex device comprises a 64 x 96 array of look-up-table based configurable logic blocks (CLBs) each of which includes four logic cells (LCs). These CLBs provide the functional elements for constructing logic. Each LC has a 4-input function generator, carry logic and a storage element (Figure 14). Two logic cells form a slice which is often used as the unit to express the hardware utilization. The function generators are implemented as 4-input Look-Up-Tables (LUTs). Each LUT can be configured as 16x1 bit synchronous RAM. Two LUTs can be combined to create a 16x2 bit or a 32x1 bit synchronous RAM, or a 16x1 bit dual port synchronous RAM [59]. The storage element can be configured either as edge triggered D flip-flops or as level sensitive latches.

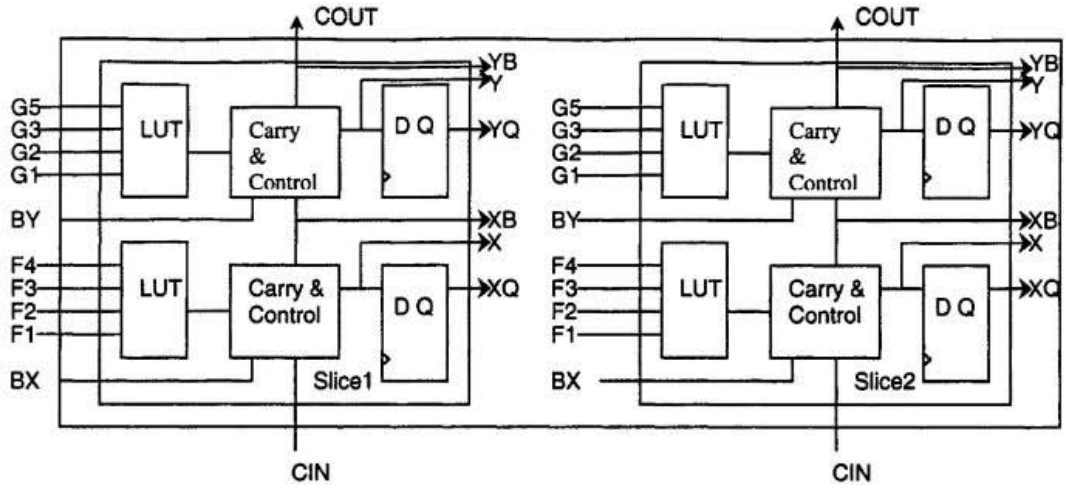


Figure 14. 2-slice Virtex CLB [59].

The XCV1000 device has a 680-pin ball grid array package, which provides 512 I/O pins and over one million system gates. The Virtex device incorporates several large block select RAMs (BRAMs), organized in columns. These complement the distributed LUT based RAM structure in CLBs. The XCV1000 has 32 block select RAM blocks. Each block is a fully synchronized dual port 4096-bit RAM. These can be configured either as RAMs or ROMs.

Virtex devices feature a flexible, regular architecture that comprises an array of CLBs surrounded by programmable input output blocks (IOBs). All these are interconnected by a rich hierarchy of fast, versatile routing resources (Figure 15). The abundance of routing resources is attractive for implementing large and complex designs. "VersaRing" facilitates pin swapping and pin locking, which are required for adopting

the existing PCB layouts when the device is redesigned [59]. Virtex device also includes four clock delay locked loops (DLLs) for advanced clock controls.

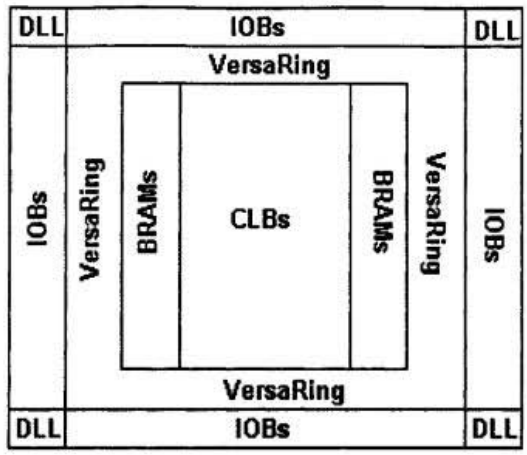


Figure 15. Virtex architecture overview [59].

3.5.2 Design Methodology

Basically two hardware design methodologies are available: Hardware Design Language (HDL) based method and schematic based method. In general schematic based designs give slightly better results in terms of area and speed compared to their HDL based counterparts. However a schematic based method is not feasible for large complex designs [51]. Hence an HDL based methodology, with Very High Speed Integrated Circuit Hardware Description Language (VHDL) as the language, was chosen. For the synthesis and implementation the tools provided by Canadian Microelectronics Company (CMC) were used.

3.5.3 Design Flow

In these implementations, the FPGA design flow given by Xilinx was utilized. Figure 16 shows the main steps of this flow.

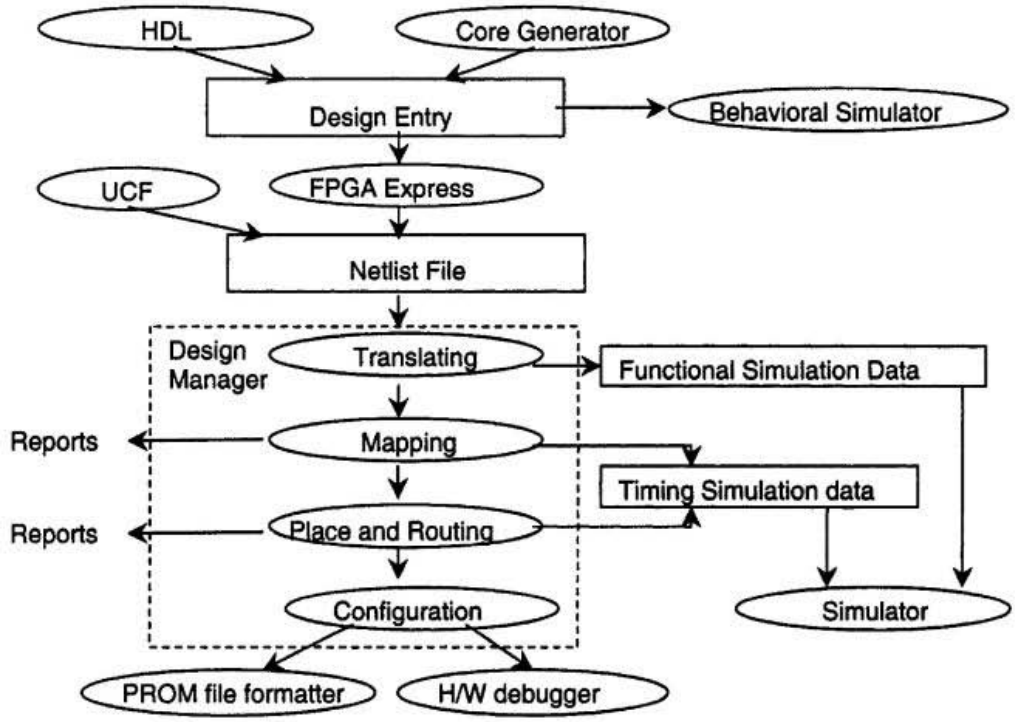


Figure 16. FPGA Design flow [60]

Initially the behavioral model of the design is implemented using VHDL. In this design there are some standard operations for which we can use the Xilinx library cores, which have already been optimized. These components can be generated using the Core Generator. This is a tool used for parameterizing cores that have been optimized for Xilinx FPGAs. The advantages of using a core based design process include shorter design cycle time, reduced risks and improved performance through higher levels of integration, among others [61]. The behavioral model is simulated for functional

correctness using Synopsys VHDL System Simulator (VSS). Then the VHDL model is synthesized to create the netlist file using FPGA Express provided by Synopsys Inc. During synthesis the design is flattened to a large number of processes that communicate via signals known as nets. The netlist file is a data structure that describes all of the components connected to each net [62].

In order to carry out the functional simulation, the design has to be translated using the Xilinx flow engine. This uses the Native Generic Database Build (NGDBuild) program to read the netlist file along with any constraints specified and create the NGD file. The NGD file contains the logical description of the design expressed both in terms of the hierarchy used when the design was first created and lower level Xilinx primitives to which the hierarchy resolves [63]. Therefore this functional simulation is also known as the post-NGDBuild simulation. If this is satisfactory the design is mapped. At this level it is possible to carry out post-map simulation, which is a gate level simulation with real gate delays and estimated delays for routing. This simulation is useful in determining if there are obvious timing issues in the design before carrying out place and route [64].

Finally, after the placed and route (PAR), the post-route simulation is carried out. This is a gate level simulation with real delays for the gates and routing. At this point the static timing analysis is carried out. Using the NGD file it is possible to create a structural VHDL file and a Standard Delay Format (SDF) file. SDF is an industry-standard format for passing back-annotated delay information to the structural HDL. With these files the timing simulation with back annotation can be carried out.

After the PAR step, the CLBs on the chip must be configured to implement the behavior of the netlist components that have been mapped, placed and routed to them. This is done by determining the values of configuration bits required to program the device. This is called bit generation. By loading these configuration bits into the FPGA, the device can be customized [62].

3.6 Hardware Architectures

The cryptographic hash functions studied in this thesis have an iterative structure based on the method proposed by Merkle and Damgard [65]. The inherent sequential nature of this structure provides only limited opportunities to enhance its performance. However several architecture options can be adopted. Loop unrolling architecture allows for unrolling of multiple steps, up to the total number of steps required by the algorithm. In this approach as the number of unrolled steps increases the hardware utilization increases but the complete algorithm processing delay decreases through hardware minimization across steps. Iterative looping is a subset of loop unrolling in which only one step is unrolled. The iterative approach in general minimizes the hardware requirement but maximizes the time requirement since it needs a large number of clock cycles to perform a hashing. By implementing partially unrolled designs it is possible to have a range of area vs. time trade offs. In this study, the following two extreme cases of these hardware architectures are investigated:

- Full loop unrolling
- Iterative looping

Full loop unrolling

This architecture allows for unrolling of multiple rounds up to the total number of steps required by the hash algorithm. In this case all the rounds are implemented as a single combinatorial block. Operations such as variable shifting can be directly implemented without any hardware such as barrel shifters. The number of required multiplexers can be reduced too. However, while this approach minimizes the time for a hash operation, it maximizes the hardware utilization.

Iterative looping

Only a generic step is implemented and it is iterated for the number of steps required for the hash operation. This approach has a low register-to-register delay but requires large number of clock cycles to perform a hash operation. In general, although it needs extra components such as multiplexers and barrel shifters, the hardware utilization is reduced.

3.7 Conclusion

In this chapter the appropriateness of FPGA implementation of cryptographic algorithms was discussed with a description of some previous studies related to AES candidate algorithms. Here, several potential advantages of cryptographic algorithms implemented using FPGAs were mentioned. Following the main issues of implementation associated with this research, the design environment and

implementation choices were discussed. The architecture of the target device and the design flow utilized for the synthesis and implementation were explained. Finally the two architectures that were adopted for the implementation were briefly introduced. In the next chapter the details of implementation of MD5, SHA-1 and HMAC-SHA-1 will be presented.

Chapter 4

Implementation of MD5, SHA-1 and HMAC-SHA-1

In this chapter, various aspects of the implementation of MD5, SHA-1 and HMAC-SHA-1 algorithms are discussed. Initially the design details of the architectures are explained and then the simulation results are discussed. The actual implementation results and some optimization techniques are elaborated thereafter. Finally the performance analysis is carried out for all the designs based on the implementation results.

The design flow given in Section 3.5.3 was used throughout the implementation process. The implementation involves several simulations at different stages. This process assures the correct functionality of the design at each stage before proceeding to the next level of the design flow.

4.1 MD5 Implementation

MD5 algorithm is a block-chained hashing algorithm. The hash for a block depends on both the block data and the hash of its preceding block. As a result, blocks cannot be hashed in parallel. Each step consists of four additions, three component logical operations, two table lookups and one rotation. The tree of operations can be optimized by performing operations, which involve items not dependent on the previous

step, early. According to Figure 7, the item that depends on the previous step is word "B" and hence the result of logical operation has a considerable delay. The optimized tree of operation (assuming each operation takes one unit time) will be as given in Figure 17. According to this one time unit step can be reduced from the times required for the general structure given in Figure 7 [66].

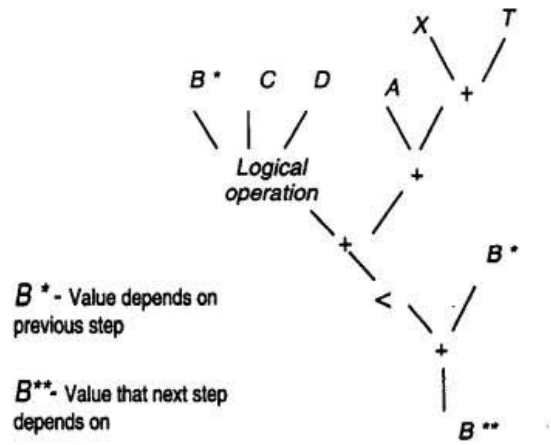


Figure 17. Optimized operation tree

As discussed in the previous chapter, two architectural options are investigated and implemented: iterative looping and full loop unrolling. Both architectures are implemented at the behavioral level in VHDL, synthesized and functionally simulated. After verifying the functionality, the design undergoes the process of the flow engine. The functionality of the PAR implementations is then re-simulated with back-annotated timing using the same test vectors used in functional simulation, thereby verifying that the implementation of the design is successful. In both designs, it is assumed that the first two phases of the algorithm have already been performed and the input of message blocks can be controlled according to the state machine states.

4.1.1 Iterative Architecture

By implementing a generic step of the MD5 algorithm, a looping architecture with 64 iterations provides the greatest area optimized solution. In this design "MD5 iterative core" is the generic step, which is shown in Figure 18. A few additional multiplexers and a barrel shifter have to be used to perform the selection of the round function and the variable shifting in each round. Among the several components, the block select RAMs, ROMs, adders, registers and multiplexers are parameterized using core generator modules of the Xilinx tool.

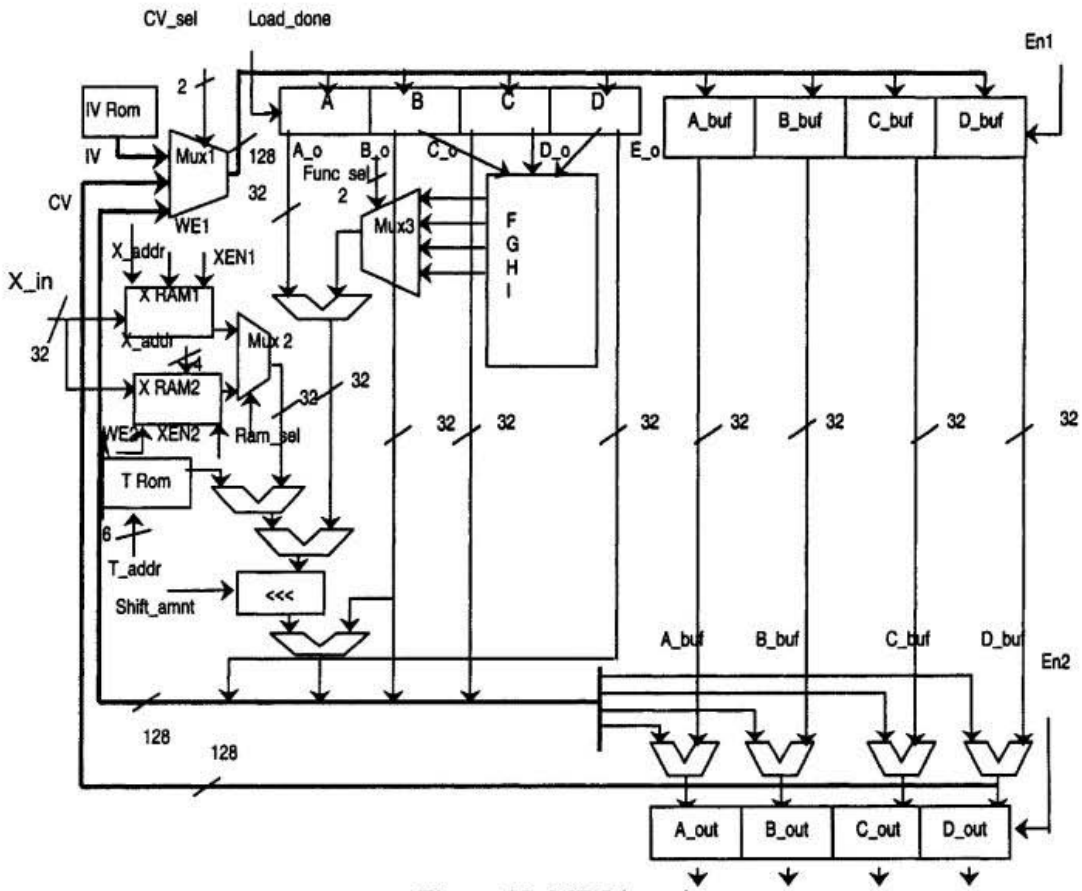


Figure 18. MD5 iterative core

The block diagram of the iterative design data path is shown in Figure 19. The message is loaded using a 32-bit bus "X_in" and the digest appears as four 32-bit words. The "X_in counter" provides the addresses (X_count) for 16 words of a message block to be written into the RAM modules. The "X_sel" signal from the "MD5 iterative state machine" provides the address for reading the appropriate message word from a RAM module. The "RAM select counter" is used to count the cycles required to switch between the two RAMs for reading and writing.

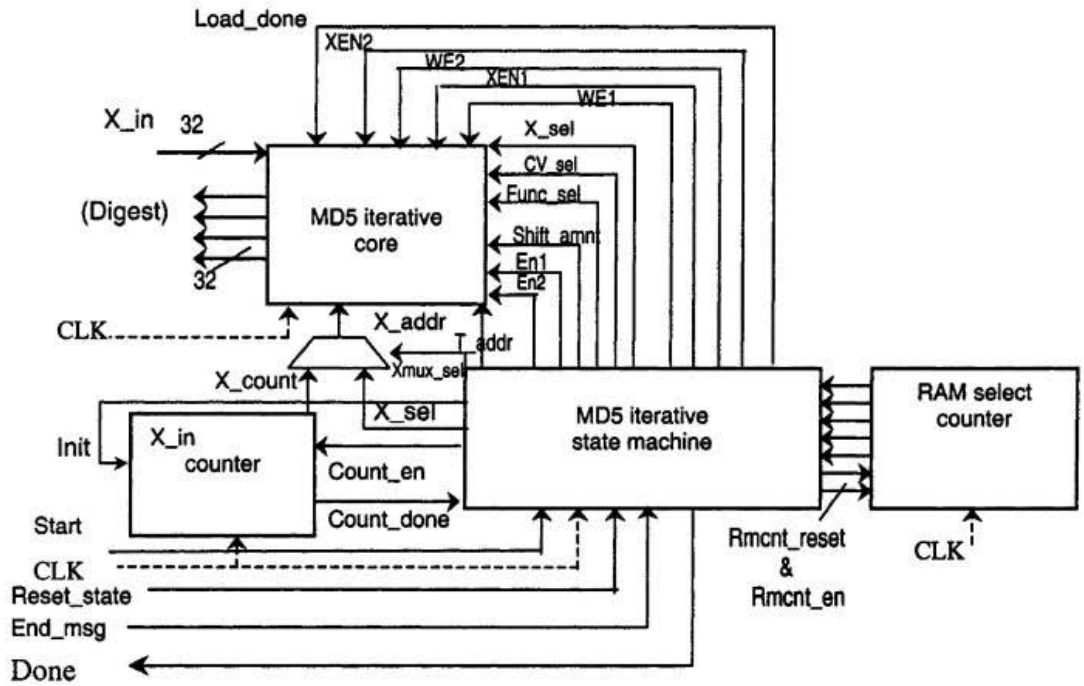


Figure 19. Block diagram of MD5 iterative design.

The "MD5 iterative state machine" provides all the control signals required for all the operations. The basic state diagram is shown in Figure 20. This has 68 states including the three states required for initializing and loading the very first block to the core. The

subsequent block operations need 65 states. An important feature of this design is the loading of message blocks in parallel with computation. The two RAMs can be utilized to load the next block while the present block is being used in computation. This eliminates the loading time from the total time to process all but the first block. The 512-bit message block is loaded to the core using a 32-bit bus.

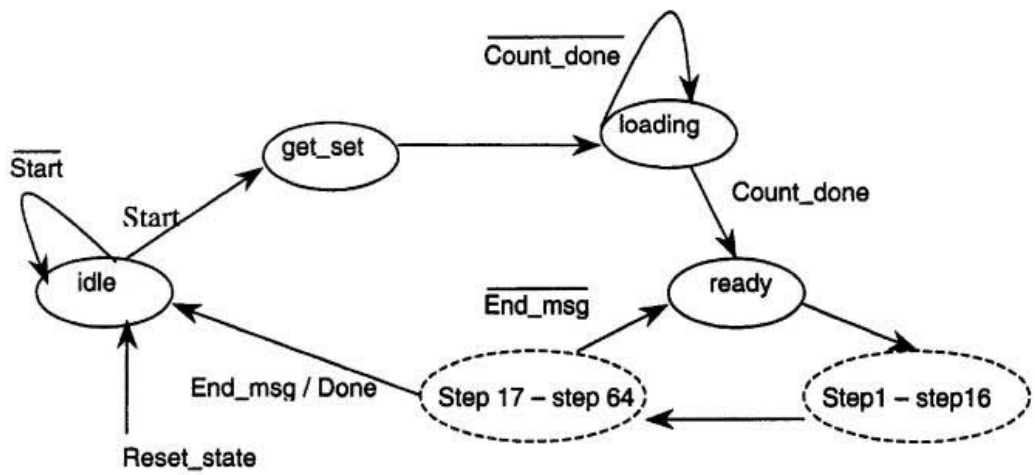


Figure 20. State diagram for MD5 iterative design

The "Reset_state" signal initiates the state machine and the counters at the "idle" state. Then with the "Start" signal the function starts with moving to "get_set" state. The initial vectors are loaded in parallel to the input register and to a buffer. The initial vectors as well as the chaining variables are kept in this buffer until the 64^{th} step to get added with the last result to form the chaining variable for the next block. Then during the "loading" state, the first block is loaded to the "XRAM1" using the addresses given by the "X_in counter". After that the state machine starts to provide addresses for reading of "XRAM1". Using the first 16 addresses provided by the state machine, the next block is

written to "XRAM2". After the 64th step, "XRAM2" is read. During the first 16 steps of processing the second block, the third block is written to "XRAM1". This reading and writing of RAMs alternates in every 64-clock cycles. Subsequent blocks utilize the previous chaining variable as their initial values. At the end of 64th step, if the "End_msg" signal is asserted, digest appears at the output as four 32-bit words. The "Done" signal indicates that the digest has been created.

4.1.2 Full Loop Unrolled Architecture

The full loop unrolled architecture has a 64-step combinational logic core as shown in Figure 21. This provides the best time-optimized solution.

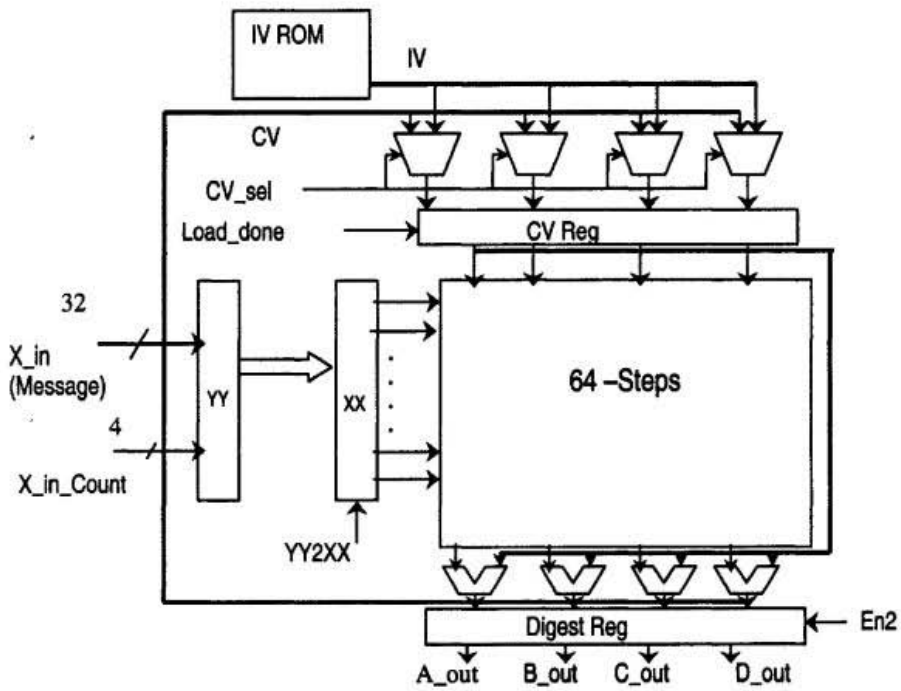


Figure 21. MD5 full loop-unrolled core

In this architecture all the elements of each step are implemented as combinational logic. The barrel shifter has been replaced by direct wiring of appropriate shifted bits in each step. As with the iterative design, the use of double buffering ("XX" and "YY") eliminates the loading time from the critical timing path. The next block is loaded during the computation of the present block. "IV ROM" provides the initialization vector for the first step. The "Load_done" signal makes the initialization vector and the chaining variables available for the first block and for the subsequent blocks respectively.

During computation of the digest for a block, the next block is stored in buffer "YY" and after the computation the "YY2XX" signal goes high and hence "XX" obtains the new input for the next computation. The block diagram of the complete data path is given in Figure 22.

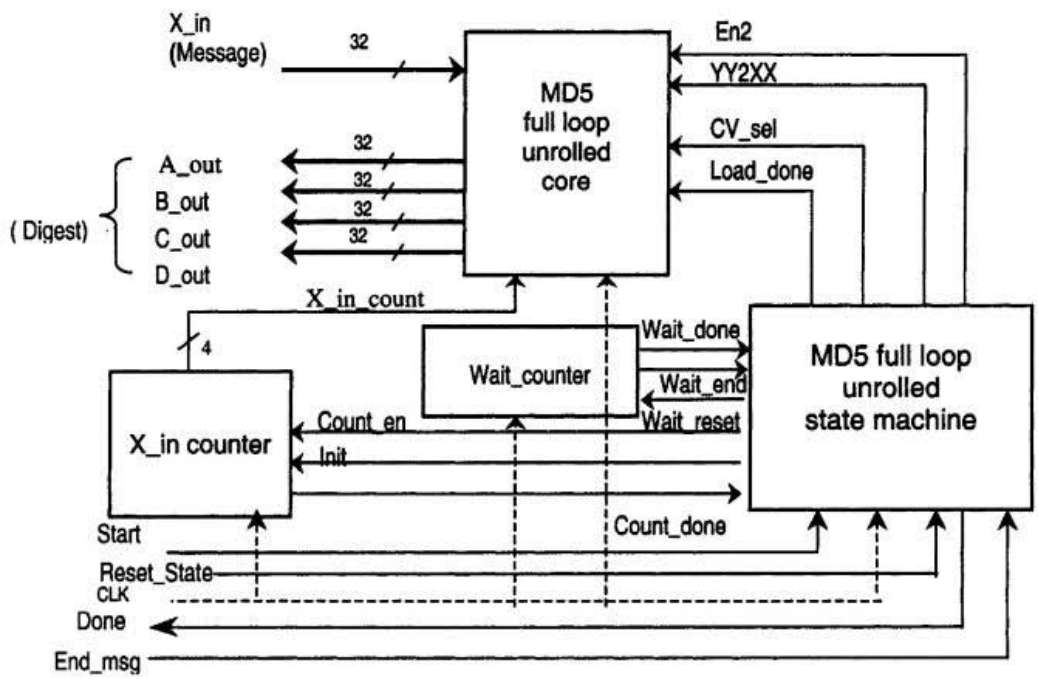


Figure 22. Block diagram of full-loop-unrolled design.

In addition to the core, the other main components are the state machine, "X_in counter" used for loading the blocks to the core and "Wait_counter" utilized to count the number of cycles for the combinational logic delay of the computation.

Similar to the iterative design, the "Reset_State" signal initiates the state machine and the "X_in counter". The initialization vectors are taken into the register "CV_Reg". With the "Start" signal, the initial block is loaded to buffer "YY" and right after that "YY2XX" signal loads it to buffer "XX" and the computation is commenced. During computation, the next block is loaded to buffer "YY". When all the blocks in the message are processed, "En2" signal makes the digest available at the output of register, "Digest_Reg".

The state machine is simpler than that of the iterative design. It has 11 states including idle initializing and loading states as shown in Figure 23.

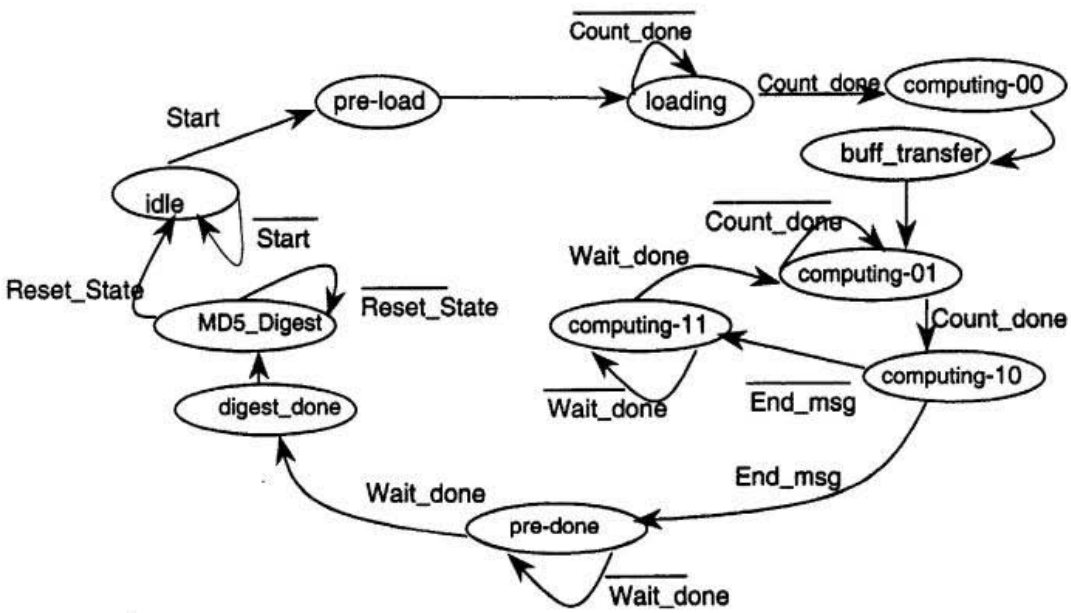


Figure 23. State machine for full loop unrolled design.

As in the iterative design, the "Reset_state" signal initiates the state machine and the counters at the "idle" state. Then with the "Start" signal the function starts by moving to "pre_load" state. The initial vectors are loaded to the input register. The first block is loaded to buffer "YY" during loading state and then it is transferred to buffer "XX" during "buff_transfer" state. In the state "computing-01" the next block is loaded to buffer "YY" while the chaining variable for the present block is being calculated. In the state "computing-10" the end of the message ("End_msg") is checked. If "End_msg" is asserted the present calculation is continued in "pre-done" state otherwise the next state will be "computing-11" state. When the state is "computing–11" the computation of chaining variables of the intermediate blocks is carried out. The chaining variable of the last block, which is the digest, is computed in "pre-done" state. Finally the "Done" signal is asserted in "digest_done" state. The output holds the digest in "MD5_digest" state until the chip is reset by "Reset_state" signal.

4.1.3 Simulation, Synthesis and Implementation Results

Iterative Design

The implementation followed the design flow described in Section 3.5.3. All the simulations were carried out using the test vectors given in [4]. Appendix A1 gives the complete simulation results. Figure 24 gives a functional simulation result. This has to be studied along with Figure 18 and Figure 19. The test message and the expected results in hexadecimal are given below. These are represented in little endian format.

Test vector:

ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz012345

6789

Expected digest (grouped in words):

D174AB98 D277D9F5 A5611C2C 9F419D9F

The 496-bit long message has to be appended with 463 zeros preceded by '1'. Then the 64-bit representation of the length of the message ("000001F0") is appended. The words of the resulting message are given below:

44434241, 48474645, 4C4B4A49, 504F4E4D, 54535251, 58575655, 62615A59, 66656463, 6A696867, 6E6D6C6B, 7271706F, 76757473, 7A797877, 33323130, 37363534, 00003938, 00000000, 00000000, 00000000, 00000000, 00000000, 00000000, 00000000, 00000000, 00000000, 00000000, 00000000, 00000000, 00000000, 000001F0, 00000000.

Hence, this message has two 512-bit blocks.

The signals shown in Figure 24 are related to those of Figure 18 as follows:

ims1:	CV_sel	ims7:	Shift_amnt	ims12:	XEN1	ims17:	En2
ims2:	Func_sel	ims8:	T_addr	ims13:	X_mux_sel	ims18:	WE2
ims3:	Init	ims9:	X_count	ims14:	X_addr	ims19:	XEN2
ims4:	En1	ims10:	Count_done	ims15:	Load_done	ims26:	Rmcnt_reset
ims6:	X_sel	ims11:	WE1	ims16:	Count_en	ims27:	Rmcnt_en

The first block of message is written to X_RAM1 using the address "X_addr" (ims14) provided by "X_in counter". During this time the "Ram_select_counter" is disabled by setting the signal "Rmcnt_reset" (ims26) high. As "Load_done" signal (ims15) becomes high the state machine starts to give the addresses for loading the next block to X_RAM2. Concurrently, the time computation of first chaining variable begins. Now the "Ram_select_counter" is enabled for alternating the RAMs for reading and writing. The signals "WE1" (ims11) and "WE2" (ims18) are asserted accordingly. Since the reading of first 16 words is in sequence, those addresses can be used for writing the next block on to the other RAM. At the end of the message the "Start" signal goes low and after the last step the four outputs, A_out, B_out, C_out and D_out hold the message digest.

The interface of the synthesized MD5 iterative design is given in Figure 25. The design was implemented using medium effort for synthesizing and PAR. Then the process was repeated with high effort. Next, according to the critical path timing details, "Period" timing constraint was introduced. Different constraint values were applied to get the best timing result.

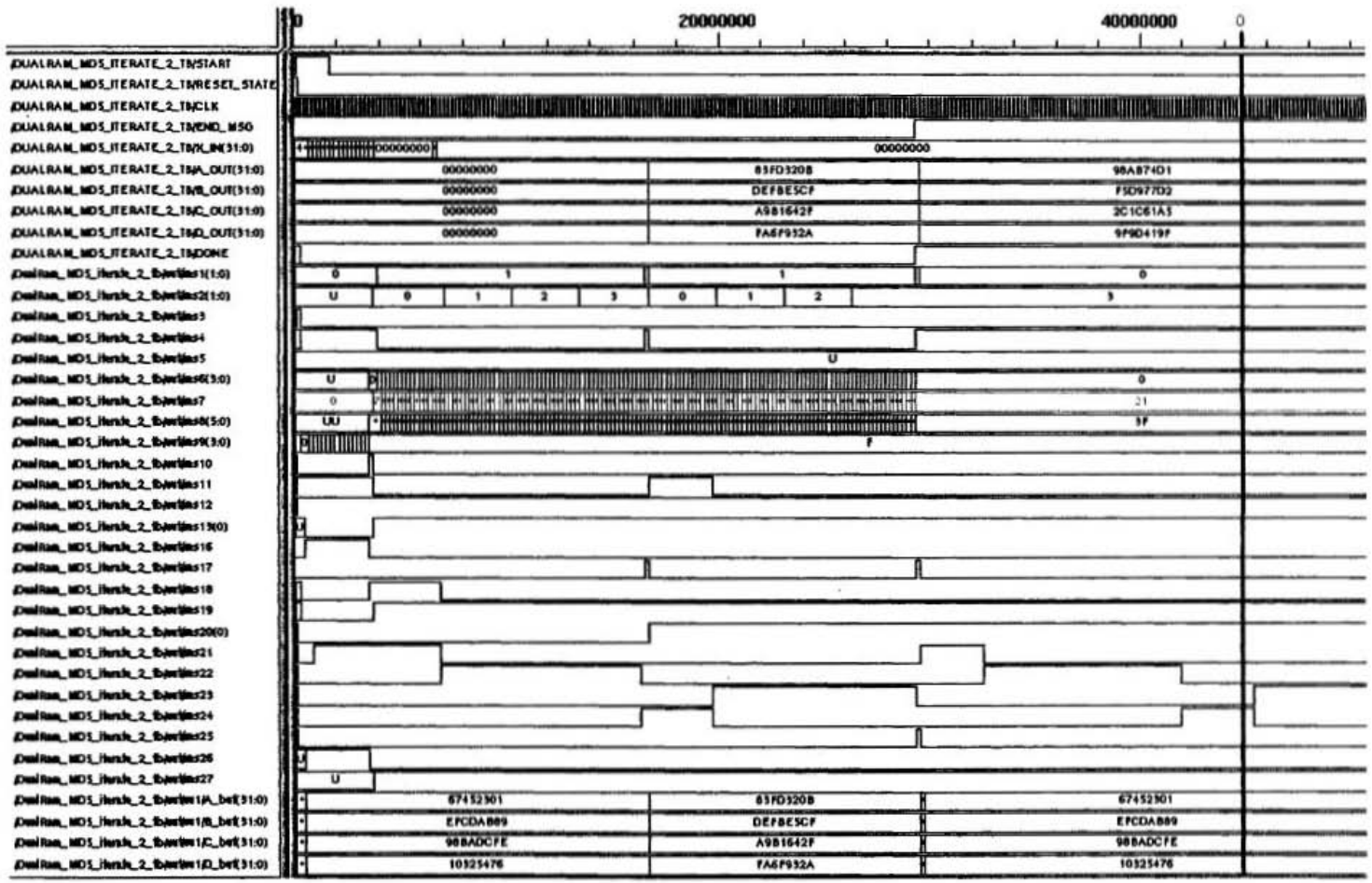


Figure 24. Functional simulation of Iterative design

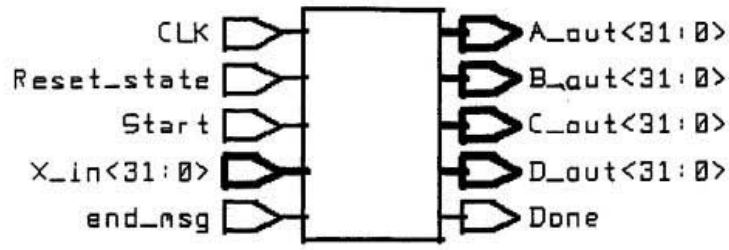


Figure 25. Interface of the MD5 iterative design

In the case of the iterative design, for all the implementation trials of different effort levels or constraints, the utilization of the external IOBs was 161 out of 512 (31%) and the block RAM usage was 2 out of 32 (6%). The number of slices used for this architecture was significantly low. It was 877 out of 12288 (7%) and from this the barrel shifter utilized 288 (2%). There is 4% utilization of three state buffers (TBUFs). The summary of timing reports is given in Table 1. In this table the critical path delay is the delay for a single iteration.

Table 1. Timing report summary of MD5 iterative design

	Medium Effort	High Effort	High Effort with a Timing Constraint
Number of slices (% of total)	7 %	7 %	7 %
Number of I/O block (% of total)	31 %	31 %	31 %
Logic delay (% of total)	25.8 %	37.7 %	37.1%
Routing delay (% of total)	74.2 %	62.3%	62.9 %
Critical path logic levels	16	33	25
Total delay of critical path (ns)	39.57	35.75	34.68

The critical path differed for different implementation trials. However, as the effort level becomes higher and the constraints get tighter, the critical path logic level has been changed. But this has been limited to the CLBs, which have already been utilized for the design during implementation with medium effort. Hence the total slice utilization has not been changed. The maximum delay of the critical path decides the frequency at which the design can operate. According to the timing simulation the maximum frequency of the design was 28.83 MHz. The performance is discussed in Section 4.4.

Full loop unrolled design

The same procedure used for the iterative deign was adopted for implementation of full loop unrolled design. The functional simulation results, for the same set of test vectors used in iterative design, are shown in Figure 27. This has to be referenced to the Figures 21 and 22. Initially the test message has to be prepared by adding the padding bits and length field as described for the iterative design. As "Reset_state" becomes low and "Start" becomes high, the initial vectors appear at the output of "CV_reg": A_o, B_o, C_o and D_o. The "X_in counter" is enabled by "Count_en" signal (cnt_en) for loading message blocks to the array "YY" during the states "loading" and "computing11". It is disabled during the other states. The address for each 32-bit word is provided by "X_in_count" signal (count). The first block is written to both "YY" and "XX" arrays, as the "YY2XX" signal (Y2X) is high during "loading" state. When it is done the "Count_done" signal (cnt_done) becomes high and the block is used for computation of the first chaining variable. The "Wait_counter" is enabled as the "Wait_reset" (wt_rst)

signal becomes low. Unlike the iterative design, the computation time in this case is a combinational delay of the critical path of the design. This delay is allocated by the "Wait_counter" by counting a fixed number of clock cycles (60 cycles in this case). The actual number of clocks has to be obtained according to the timing simulation results after the design is place and routed. After the allocated time "Wait_done" (wt_done) signal becomes high and the first chaining variable is available at the output as well as at the output of the "CV_reg". Then "YY2XX" signal (Y2X) becomes high and the next block is loaded to the array "XX" and it becomes available to the core for computation. After the second computation the last chaining variable appears as four 32-bit words: A_out, B_out, C_out and D_out. This is the expected digest in the little endian format: "D174AB98D277D9F5A5611C2C9F419D9F".

The hardware interface of the synthesized full loop unrolled design is shown in Figure 26. As in the case of the iterative design, here also for all the three-implementation trials the utilization of the external IOBs was 162 out of 512 (31%). The number of slices used for this architecture was 4838 out of 12288 (39 %). There was 2% utilization of three state buffers (TBUFs).

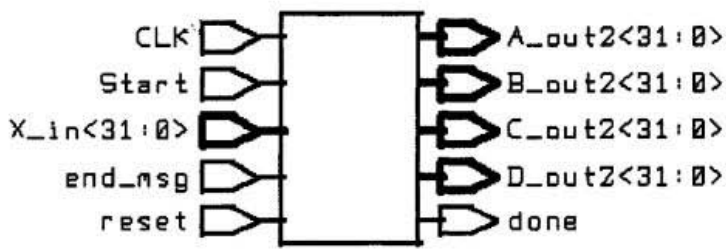


Figure 26. Interface of the MD5 full loop unrolled design

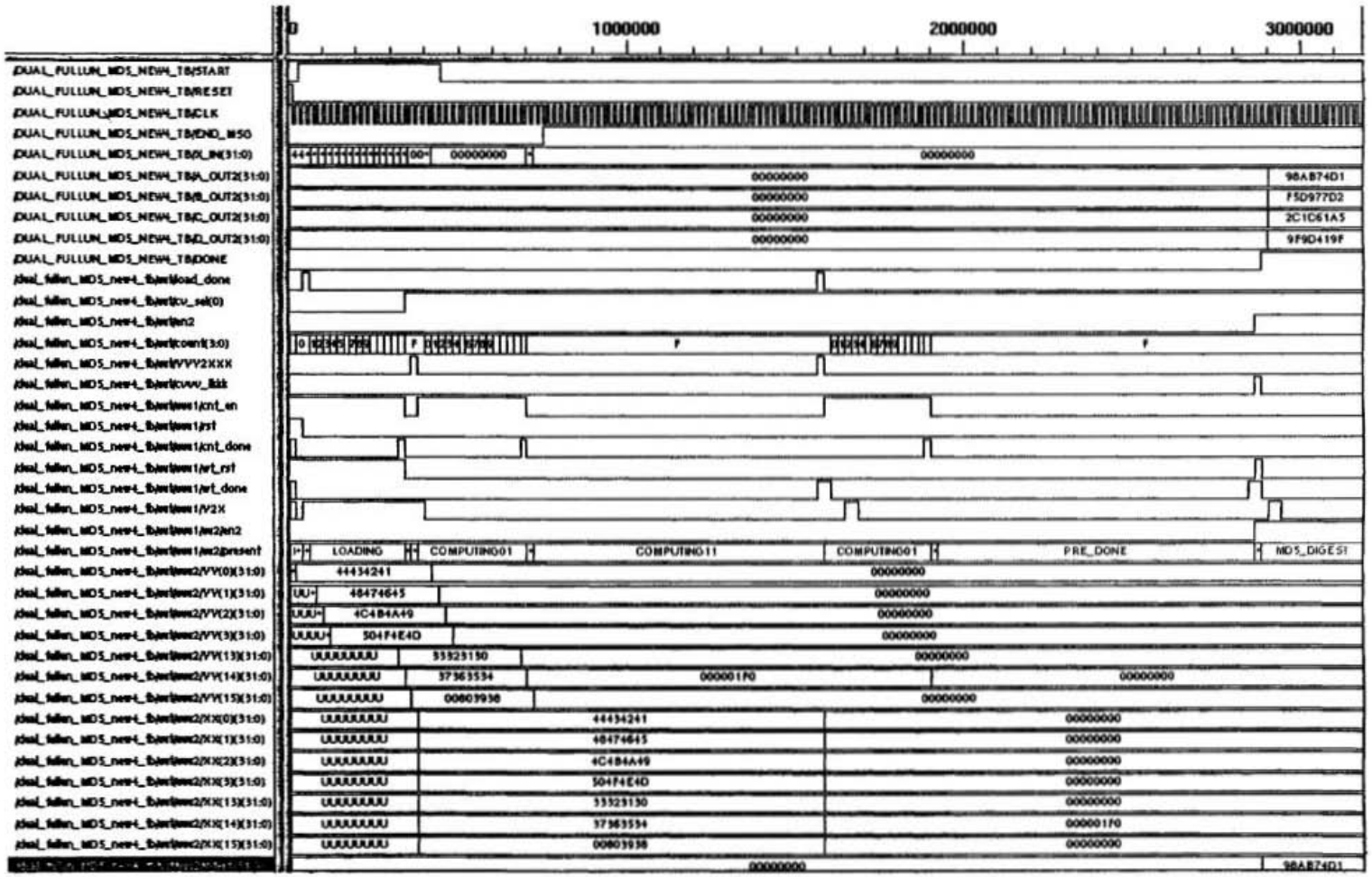


Figure 27. Functional simulation of full loop unrolled design

The summary of timing reports is given in Table 2. In this table the critical path delay is the delay for all the 64 steps.

Table 2. Timing report summary of MD5 full loop unrolled design

	Medium Effort	High Effort	High Effort with a Timing Constraint
Number of slices (% of total)	39 %	39 %	39 %
Number of I/O block (% of total)	31 %	31 %	31 %
Logic delay (% of total)	37.8 %	40.7 %	43.6 %
Routing delay (% of total)	62.2 %	59.3 %	56.4 %
Critical path logic levels	857	860	882
Total delay in the critical path (ns)	1195.90	1121.01	1054.12

According to Table 2, the optimization has improved the timing performance by approximately 12 %. Although the logic level in the critical path has been increased, there has been a reduction in the routing delay. Due to the combinational nature of this design, the block select RAMs could not be used for memory modules. The constructs used to build the RAMs and ROMs have been configured using the LUTs. The possibility of optimizing using high levels of efforts and timing constraints has been limited owing to these factors.

The maximum clock frequency at which the design can operate does not depend on the delay of the critical path, as it is purely combinational. Only the loading time of the very first block into the design core depends on this clock frequency. It has to be decided by the maximum clock rate at which the controller can generate the signals

without any set up violations. For this purpose the timing simulation was done for the controller separately and the minimum clock period was found to be 14 ns. This will be discussed later, under the performance analysis. The timing simulation at this frequency is shown in Appendix A2.

4.2 SHA-1 Implementation

The structure of SHA-1 is based on that of the MD4 algorithm. Hence many of the same modules used for MD5 can be adopted for SHA-1 as well.

4.2.1 Iterative Architecture

As given in Section 2.2.5, for SHA-1, the words processed in each step are derived from the words of the block being processed. The initial 16 words, which are used for the first 16 steps are directly obtained from the incoming message block. While they are being processed, the next words (17^{th} word and onwards) can be calculated by XORing four words from the previous 16 words together. In this case four words have to be read at a time and the calculated word has to be stored in a suitable RAM to prevent any clash between read and write operations among the modules in future steps. This can be achieved by using eight RAM modules each of size 512 bits. The arrangement of the RAM modules (RAM setup), which is an important part of the iterative design, is given in Figure 28.

The initial 16 words are written into eight RAMs during the loading period of the block. Thereafter four RAMs are read at a time and t^{th} word ($t \geq 17$), W_t is calculated while processing the first step using following relationship:

$$W_t = S^1 (W_{t-16} \oplus W_{t-14} \oplus W_{t-8} \oplus W_{t-3})$$

where, S^1 is circular left shift of 32- bit argument by 1 bit. This is repeated until all the 80 words are available for computation.

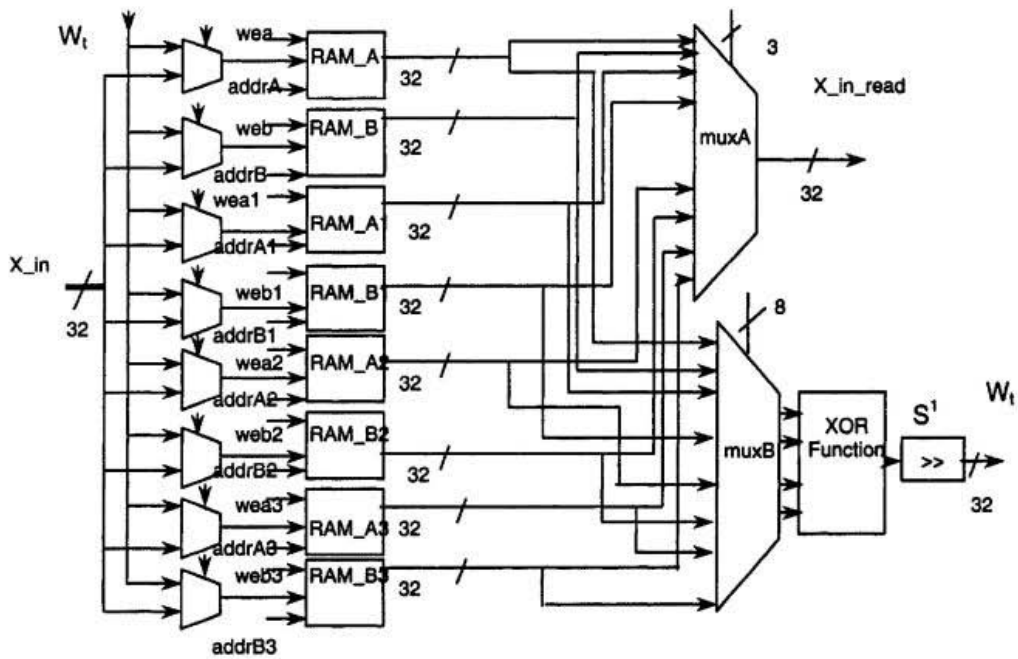


Figure 28. RAM setup for creating 80 words.

The calculated word is written in a RAM, which will not have to be read and written at the same time in a future step. The read and write process of the RAMs are given in Appendix B. Using this RAM arrangement each step obtains the corresponding block

word for that step. The iterative core of the design was constructed as in Figure 29. As in MD5, the adders, multiplexers, registers, RAMs and ROMs were generated by parameterizing the core generator modules. SHA-1 has no variable shifting in its compression function but has a fixed circular rotation. This can be easily implemented by direct wiring of the shifted bits. SHA-1 uses only four distinct constants, which are stored in a small 4-word ROM, "T_ROM".

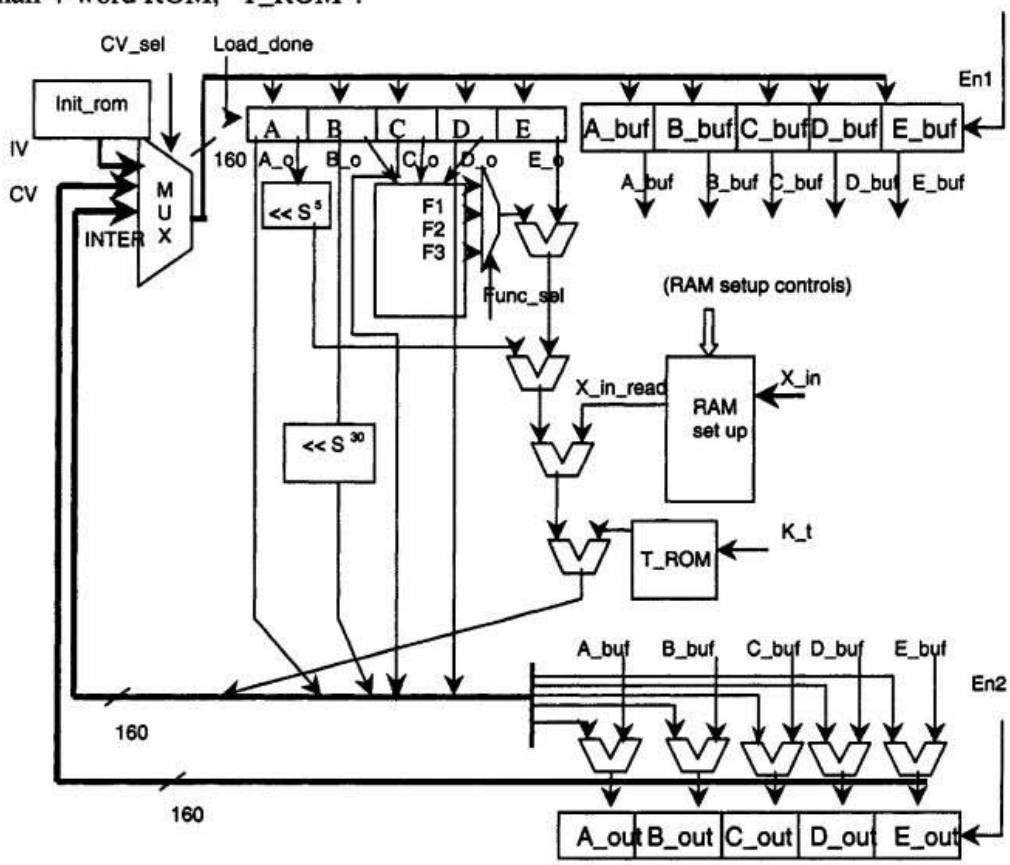


Figure 29. SHA-1 iterative core

As mentioned earlier, an iterative looping architecture provides the most area-optimized solution where a few multiplexers and RAMs are the additional hardware components. The block diagram of the complete iterative SHA-1 is given in Figure 30.

The main components are the "SHA_counter", the "SHA_1 Iterative Core" and the "State Machine" with 84 steps.

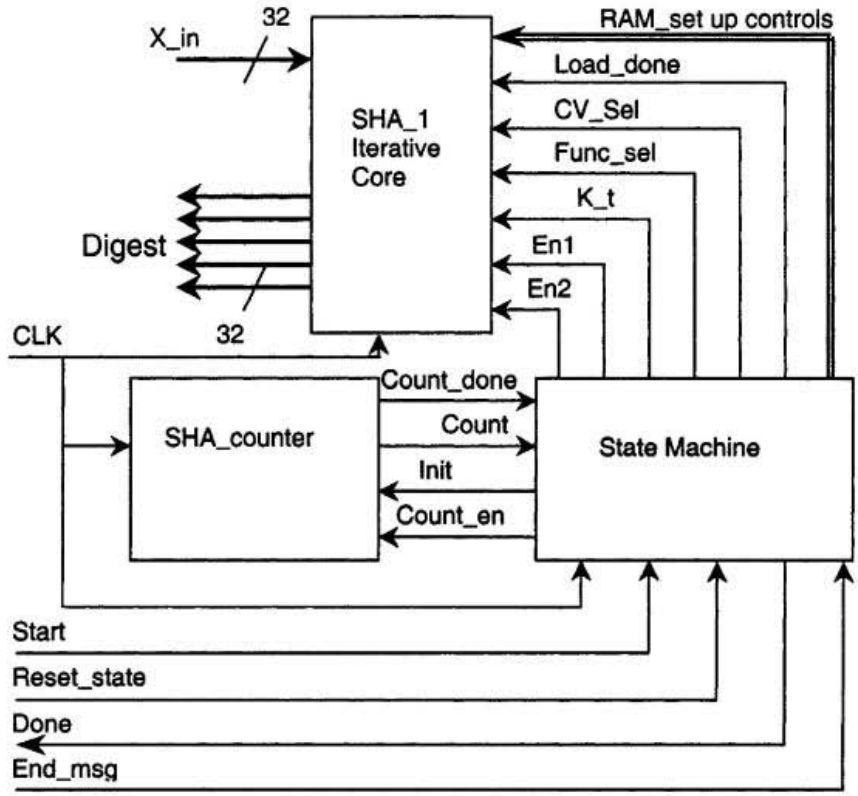


Figure 30. Block diagram of iterative design

The 512-bit message block is loaded to the core using a 32-bit bus. While the "Reset_state" signal initiates the state machine and the counters, the "Start" signal begins the operation. The initial vectors are loaded in parallel to the input registers (A... E) and to the buffer (A_buf,...E_buf). The initial vectors as well as the chaining variables are kept in the buffer until the 80^{th} step to get added with the last result to form the chaining variable for the next block. Initially the first block is loaded to the "RAM set up" using

the addresses given by the state machine. Then a particular RAM module is read to obtain the appropriate word for each step using the address given by the state machine. Subsequent blocks utilize the previous chaining variable as their initial values. The digest output comes as five 32-bit words.

4.2.2 Full Loop Unrolled Architecture

This architecture is similar to the full loop unrolled architecture of MD5. In full loop unrolled architecture, the core has an 80-step combinational logic block as shown in Figure 31. All the elements of each step are implemented as combinational logic.

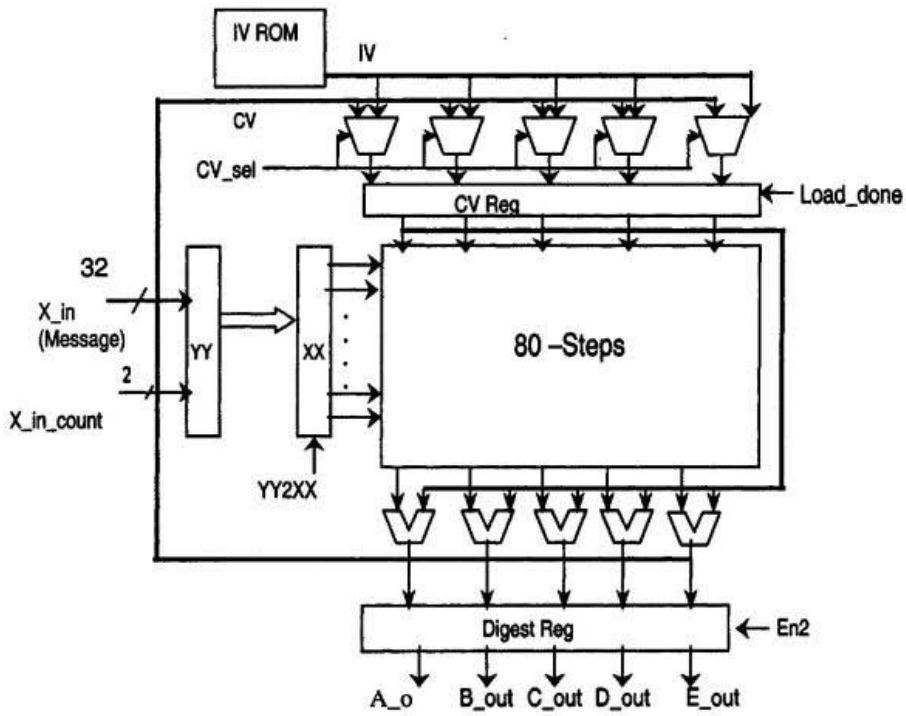


Figure 31. SHA-1 full loop unrolled core

In this architecture all the elements of each step are implemented as combinational logic. Again, the use of double buffering ("XX" and "YY") eliminates the loading time from

the critical timing path. The next block is loaded during the computation of the current block. "IV ROM" provides the initialization vector for the first step. The "load_done" signal makes the initialization vector and the chaining variables available for the first block and for the subsequent blocks respectively. During computation of the digest for a block, the next block is stored in buffer "YY" and after the computation the "YY2XX" signal goes high, hence the array "XX" obtains the new input for the next computation. The block diagram of the complete design is given in Figure 32.

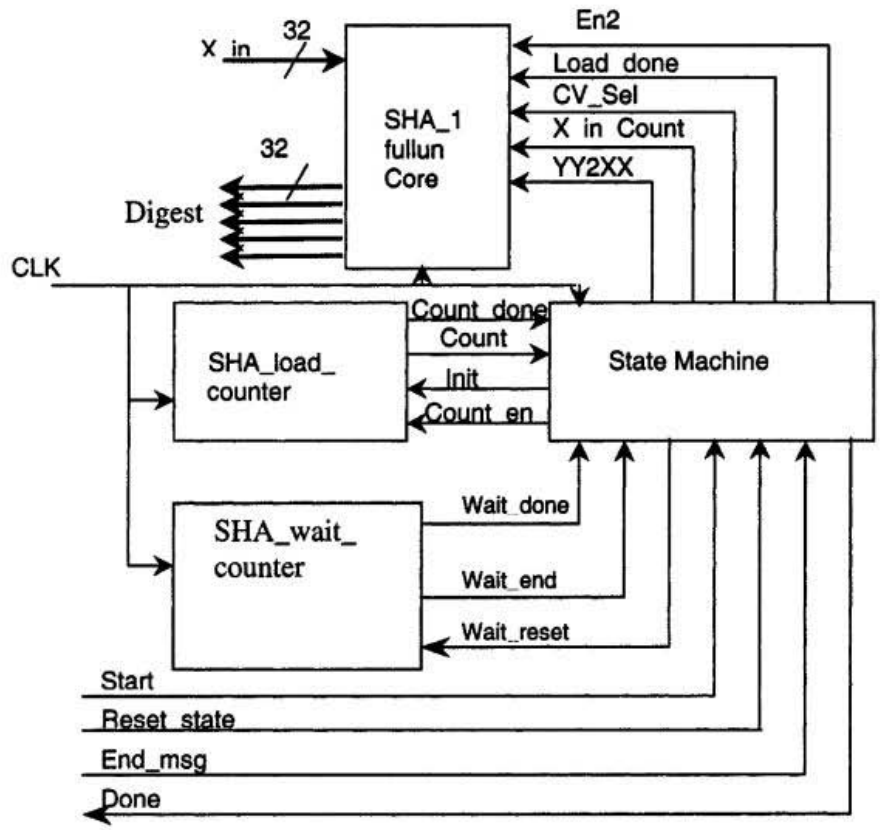


Figure 32. Block diagram of SHA-1 full loop unrolled design

The state machine has eleven states. Further, the design has two counters, "SHA_load_counter" and "SHA_wait_counter". The former is used for loading the blocks to the core and the latter is utilized to count the number of cycles for the combinational logic delay of the computation. The "Reset_State" signal initiates the state machine and the "SHA_load_counter". The initialization vectors are taken into the register "CV_Reg". With the "Start" signal, the initial block is loaded to buffer "YY" and then "YY2XX" signal loads it to buffer "XX" and the computation is commenced. During computation, the next block is loaded to buffer "YY". When the state machine observes the "End_msg" signal as high, all the blocks of the message are processed. Then "En2" signal makes the digest available at the output of register, "Digest_Reg". The "Done" signal indicates the completion of the computation.

4.2.3 Simulation, Synthesis and Implementation

Both architectures were behaviorally simulated and then synthesized. The design was initially synthesized and implemented with medium effort. The process was repeated several times with high efforts and with timing constraints. During implementation, functional simulation was carried out and then timing simulation was done with back annotation following synthesis. All the simulations were carried out using the test vectors provided in secure hash standard [8].

Iterative Architecture

The simulations were carried out using following test suite. All the values are in hexadecimal. The 32-bit values are stored in registers in big-endian format.

Test vector: abc

Expected digest (grouped in words): A9993E36 4706816A BA3E2571

7850C26C 9CD0D89D

The ASCII binary code of the test vector is "0110001 01100010 01100011" (61 62 63 in hexadecimal) and the message length is 24 bits. According to the algorithm, initially "1" is appended and then 423 "0"s are appended. Then the 64-bit representation of message, "00000000 00000018" is appended at the end. The resulting block is as follows:

61626380 00000000 00000000 00000000 00000000 00000000 00000000

00000000 00000000 00000000 00000000 00000000 00000000 00000000

00000000 00000018.

The simulation was performed using a 20 ns clock period. Figure 33 shows that, at the end of "loading" state the signal "En1" makes the initial value available for step 1. Concurrently it is stored in the four 32-bit buffers "A_buf", "B_buf", "C_buf", "D_buf" and "E_buf". During the loading time, all the 8 RAMs are enabled for writing by exerting write enable signals (wea, wea1, etc). The addresses are taken from the counter through the state machine. During the 80 steps reading from and writing into each memory module are done in alternative manner. The primitive function for each round is selected by the signal "Func_sel". The RAM setup controls (Figure 29, 30) make the appropriate word "x_in_read" available at the RAM setup output for each step. At the end of 80^{th} step "En2" is asserted and the output is available. In this design double buffering of data loading has not been employed. However this could be achieved by employing several methods. One of the methods is to have two identical RAM setups and alternating them

for different blocks using a counter. Another method is to re-use the memory locations of the RAM setup after the contents are used and by carefully managing the read and write operations.

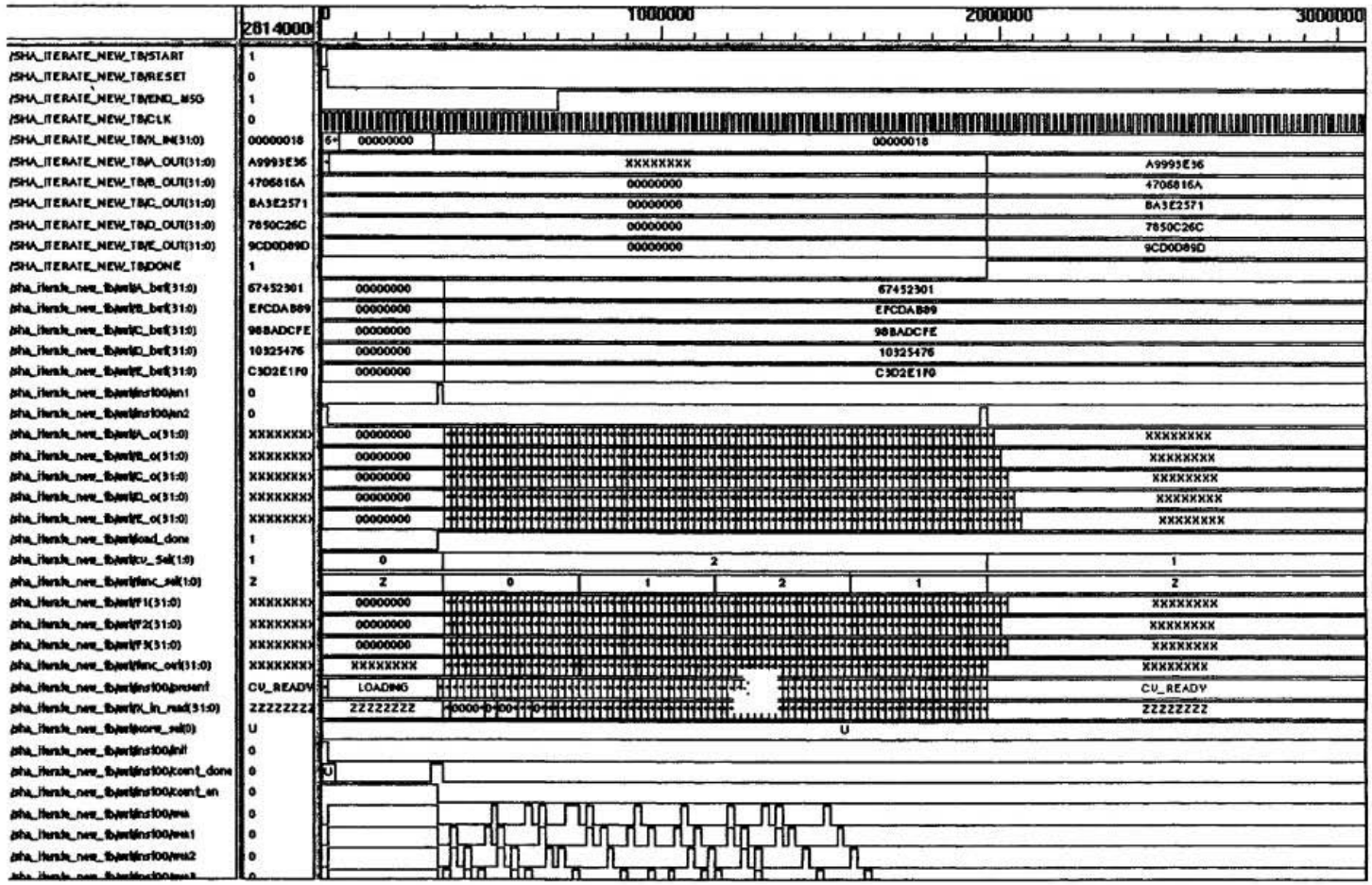


Figure 33. Functional simulation of SHA-1 iterative design.

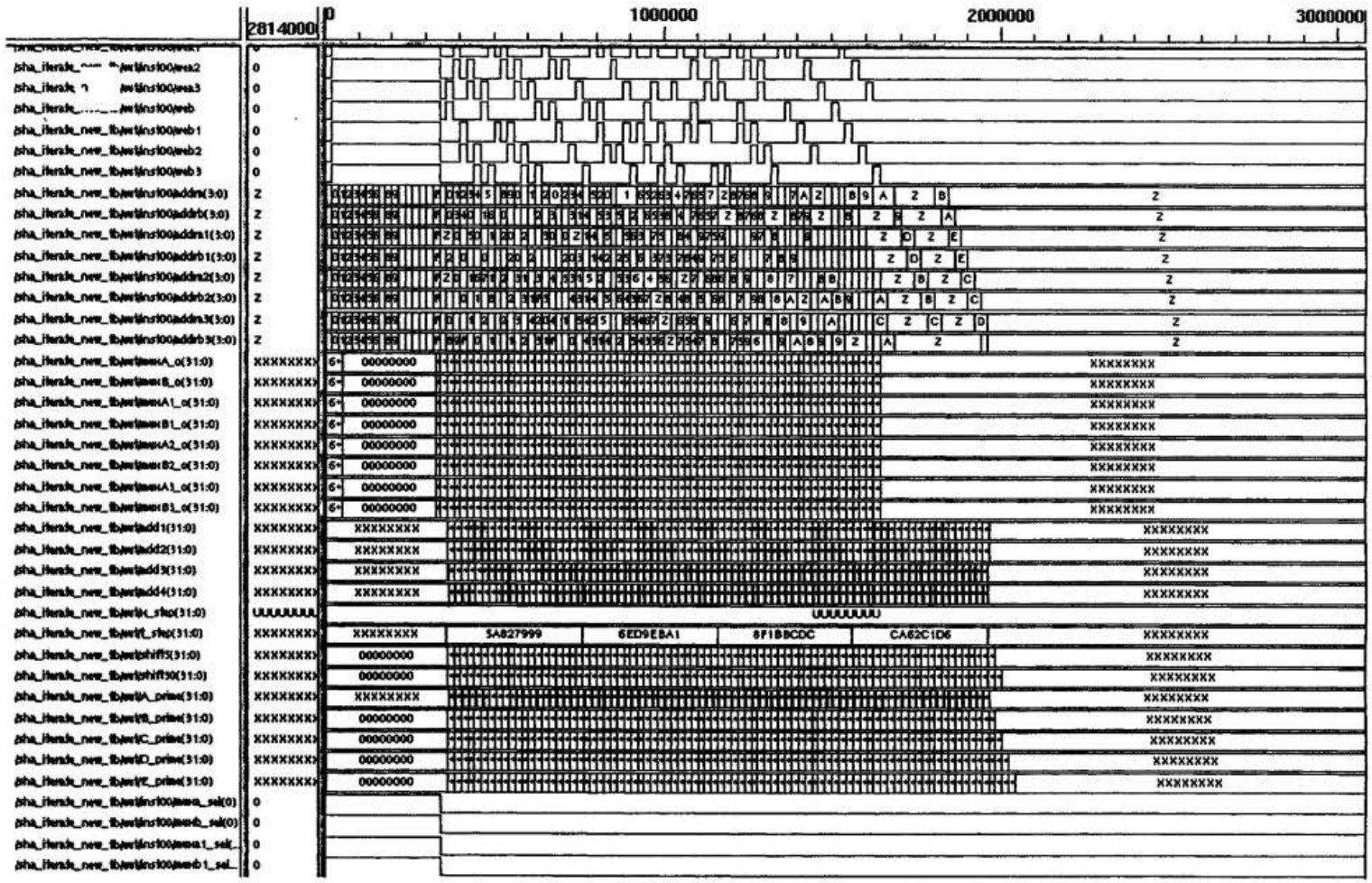


Figure 33. Functional simulation of SHA-1 iterative design (contd).

The interface of synthesized SHA-1 design is shown in Figure 34.

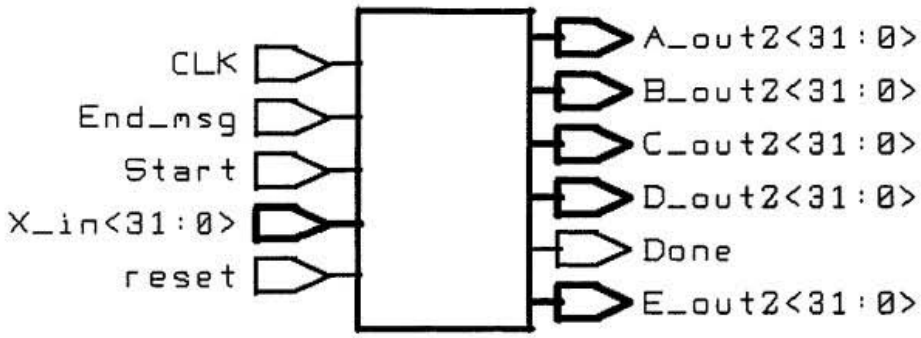


Figure 34. Interface of SHA-1 design

For the three-implementation trials shown in Table 3, utilization of external IOBs was 194 out of 512 (37 %). The number of slices used for the above architecture was 1446 out of 12288 (11 %). Furthermore there was 10% utilization of three state buffers (TBUFs). Timing reports of the SHA-1 iterative design are summarized in Table 3. In this table the critical path delay is the delay for a single iteration.

Table 3. Timing report summary of SHA-1 iterative design

	Medium Effort	High Effort	High Effort with a Timing Constraint
Number of slices (% of total)	11 %	11 %	11 %
Number of I/O block (% of total)	37 %	37 %	37 %
Logic delay (% of Total)	17.9	28.2	37.1
Routing delay (% of total)	82.1	71.8	62.9
Critical path logic levels	24	32	40
Total delay in critical path (ns)	70.917	63.24	52.996

The routing delay in the critical path has been reduced by 23 % with the use of high effort and period timing constraint. The use of many core modules and avoiding components such as barrel shifter are among the reasons for the above achievement.

Full Loop Unrolled Architecture

For the simulation, following test suite was used. The test vector is a collection of English letters and the digest is in hexadecimal.

Test vector: abcdbcdecdefdefgefghfghighijhijkijkljklmklmnlmnomnopnopq

Expected digest: 84983E44 1C3BD26E BAAE4AA1 F95129E5 E54670F1

According to the algorithm the padded message in words is:

61626364 62636465 63646566 64656667 65666768 66676869 6768696A 68696A6B 696A6B6C 6A6B6C6D 6B6C6D6E 6C6D6E6F 6D6E6F70 6E6F7071 80000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 000001C0.

The functional simulation results of full loop unrolled architecture are shown in Figure 35. Most of the signal operations are similar to those of the MD5 full loop unrolled design. The signal "YYYtoXXX" transfers the loaded data from array "YY" to "XX". At

the end of computation the signal "IMMM1" which indicates the "Load_done" signal makes the chaining variable available at A_o, B_o, C_o, D_o and E_o for the computation of second block. When the "End_msg" is observed the last chaining variable is appeared at the output with the asserted "Done" signal. The implementation results are summarized in Table 4. The slice utilization was 6963 out of 12288 (56%) and IOB utilization was 194 out of 512 (37 %). There was only 2% use of three state buffers. The timing simulation showed that the minimum clock period of the design was 25 ns. As in the MD5 full loop unrolled design, only the loading delay of the first block is affected by this clock frequency. The loading and computing times are analyzed under performance analysis in Section 4.3. In this table the critical path delay is the delay for all the 80 steps.

Table 4. Timing report summary of SHA-1 full loop unrolled design

	Medium Effort	High Effort	High Effort with a Timing Constraint
Number of slices (% of total)	56 %	56 %	56 %
Number of I/O block (% of total)	37 %	37 %	37 %
Logic delay (% of total)	41	46	51.8
Routing delay (% of total)	59	54	48.2
Critical path logic levels	1136	1263	1337
Total delay in critical path (ns)	1097.3	1009.8	906.2

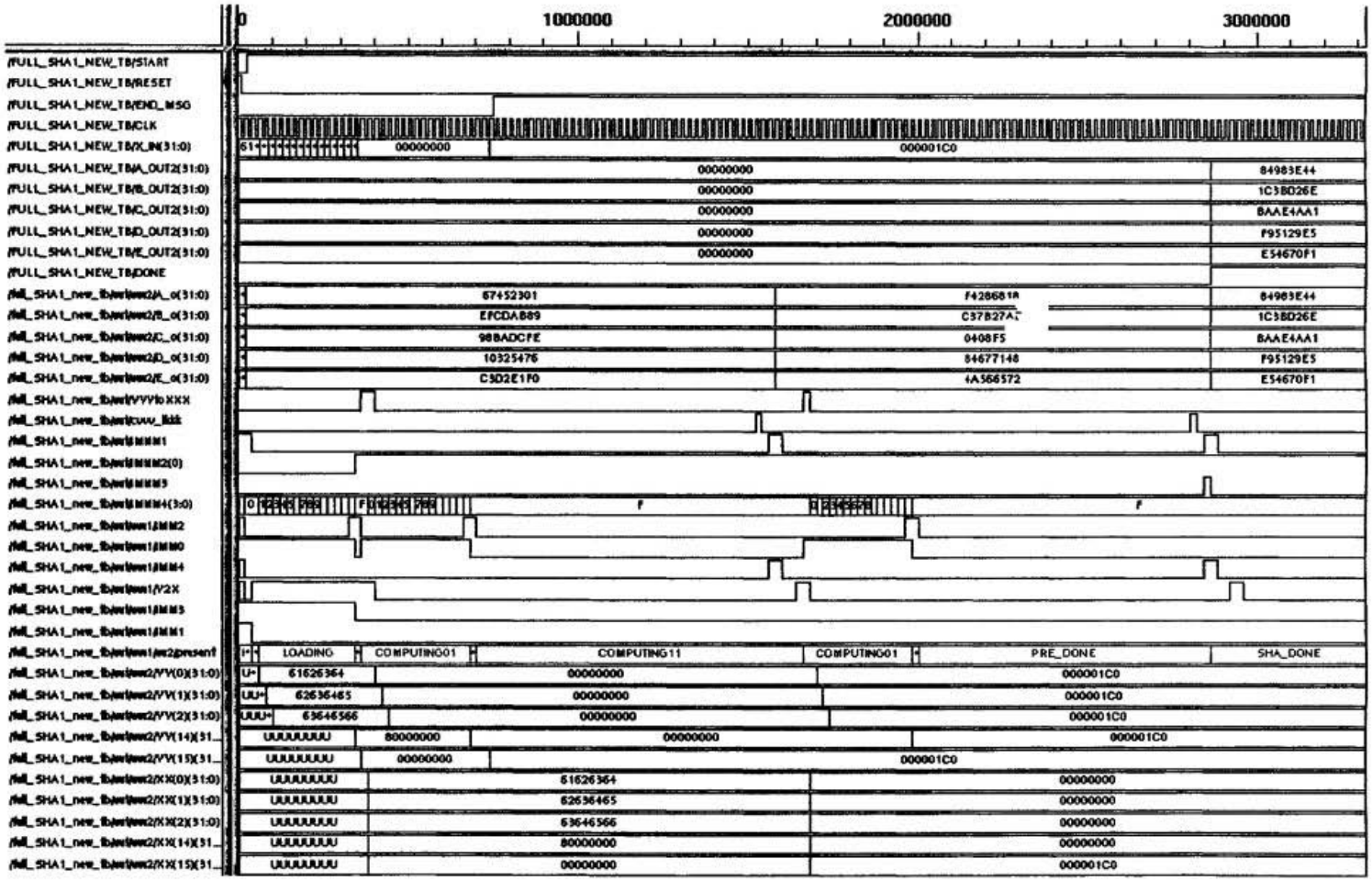
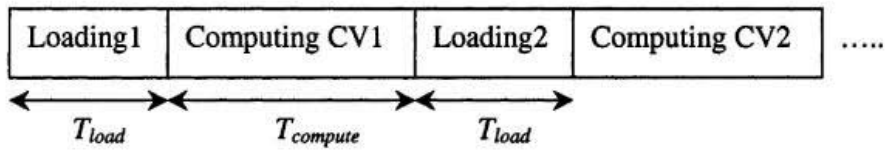


Figure 35. Functional simulation of SHA-1 full loop unrolled design.

4.3 Performance Analysis of MD5 and SHA-1

The timing analysis and throughput calculation of the three designs can be performed as follows. The two cases with and without the message loading delay are illustrated in Figure 36 and 37.

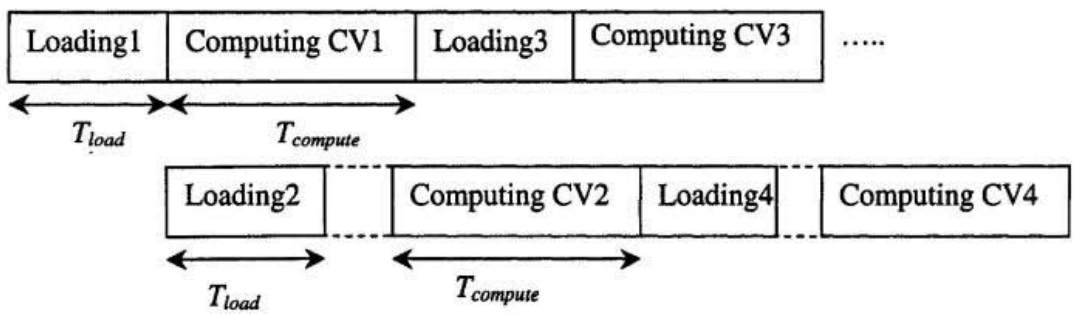
With loading delay:



$$Throughput = (512)/ (T_{compute} + T_{load}) \text{ bits/sec}$$

Figure 36. Timing diagram with loading delay

Without loading delay (double buffering data):



$$Throughput \approx (512)/ T_{compute} \text{ bits/sec}$$

Figure 37. Timing diagram without loading delay

We now discuss the results each design with respect to the timing reports.

MD5 iterative design

The use of two 512-bit RAM modules to double buffer data avoids loading delay. The minimum clock period obtained was 34.68 ns (from Table 1). So the loading time and computation time can be found as follows:

T_{load} = 16 x 34.68 ns = 554.88 ns

$T_{compute}$ = 64 x 34.68 ns = 2219.5 ns

The expected throughput (from Figure 37) is = (512) /(2219.5 ns)= 230.68 Mbps. In MD5 iterative design the critical path minimum delay is the minimum delay of the clock. Without the use of dual RAMs the throughput would be 184.5 Mbps (from Figure 36). It was evident that the use of dual RAMs leads to almost 25% improvement of the timing performance. The area utilization of this design was considerably smaller (11% of slices). The unused area could be used to implement several designs in the device and process multiple messages in parallel as far as the I/O ports can be utilized successfully.

MD5 full loop unrolled design

For this design the minimum clock period obtained was 14 ns. The minimum delay of the critical path = 1054.12 ns (from Table 2). The loading and computation times are,

T_{load} = 16 x 14 ns = 224 ns

$T_{compute}$ = 1054.12 ns.

The throughput with double buffering (from Figure 37) is (512)/(1054.12 ns) = 485.7 Mbps. Without the use of double buffering the throughput would be 400.5 Mbps. The critical delay of this design is the computational time, which is independent of the clock

period. The use of double buffering has given 21% improvement. In this case the slice utilization was 39%. Hence it is possible to have two design modules in the same device and process two different messages in parallel.

SHA-1 iterative design

In our design the loading time has not been eliminated. The minimum clock period was 52.996 ns (from Table 3). Hence the loading and computation times are,

T_{load} = 16 x 52.996 ns = 847.936 ns

$T_{compute}$ = 80 x 52.996 ns = 4239.68ns

The expected throughput (from Figure 36) is (512) /(847.936 + 4239.68ns) = 100.636 Mbps. If the loading delay is eliminated the throughput would be around 120.764 Mbps, which is a 20 % improvement.

SHA-1 full loop unrolled design.

The minimum clock period obtained was 25 ns and the minimum delay of the critical path was 906.284 ns (from Table 4). Hence the loading and computation times are,

T_{load} = 16 x 25 ns = 400 ns

$T_{compute}$ = 906.284 ns.

The throughput (from Figure 37) is (512)/(906.284 ns) = 565 Mbps. It would be 391.95 Mbps with the loading delay (from Figure 36). So the double buffering has caused 44% improvement in throughput, which is significant compared that of the MD5 design. When the two full loop unrolled designs are compared, the SHA-1 design showed better results.

The percentage routing delay of the critical path of full loop unrolled MD5 and SHA-1 were 56.4% and 48.2 % respectively. The device utilization of this design is fairly high (56% slices). It is not possible to accommodate more than one design in the same device.

The ability to reduce the routing delay has become a major reason for better performance of SHA-1 over MD5. In MD5 full loop unrolled design, the operations of each step need a unique 32-bit constant that has to be read from a ROM. This ROM that consists of 64 number of 32-bit words is configured with LUTs. The routing associated with this operation is tighter than that of SHA-1, which has only four distinct constants. Therefore the full loop unrolled design of SHA-1 could be optimized better than MD5. However, in iterative design the percentages of routing delays of MD5 and SHA-1 were almost the same. Total critical path delay of the iterative core of SHA-1 was higher than that of MD5. This is due to the larger number of logic levels in the iterative core of SHA-1 compared to MD5. Therefore in iterative designs, MD5 has better timing than SHA-1.

4.4 HMAC-SHA-1 Implementation

HMAC is a keyed message authentication code, which shall be used in combination with cryptographic hash functions specified in a Federal Information Processing standard (FIPS) [8]. Let H be the hash function initialized with its fixed Initial Value, IV, which results in an n-bit hash value. The function HMAC works on inputs m of arbitrary length, which is a multiple of 512 bits. It uses a single random string K as its key. If the key length is greater than 512, it is input to the hash function to produce an n-bit key. The recommended key length is $\geq n$. The HMAC can be expressed as follows:

$$\text{HMAC}_k(m) = H(K^+ \oplus opad, H(K^+ \oplus ipad, m))$$

where K^+ is K, padded with zeros on the left so that the result is 512 bits long, *ipad* is the inner pad, which is the byte hex 36 repeated 64 (512/8) times and *opad* is the outer pad, which is the byte hex 5C repeated 64 times. The symbol $\oplus$ denotes the XOR operation. Let the input $m = m_0, m_1 \ldots m_{l-1}$. The HMAC block diagram is shown in Figure 38 [58][16].

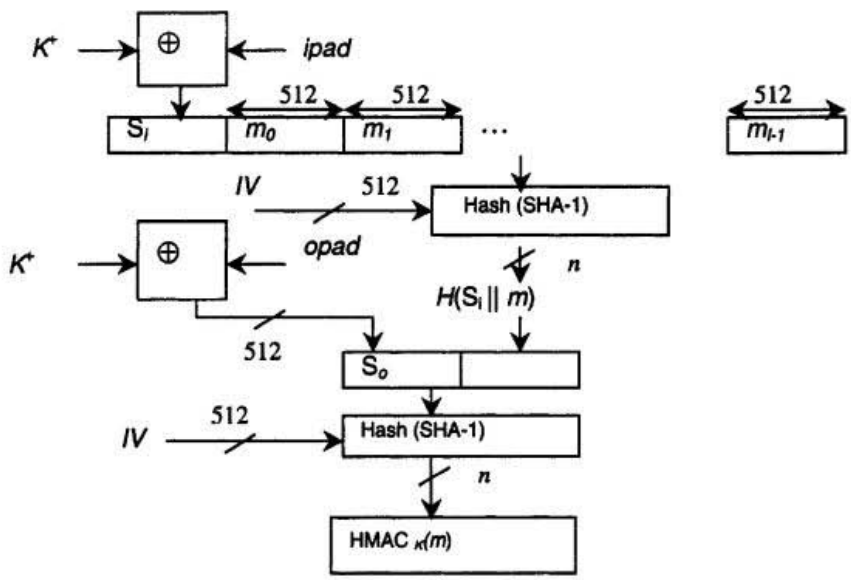


Figure 38. HMAC operations

HMAC was developed to fulfill the efficiency and other requirements of the Internet along with a more rigorous analysis of security. In HMAC [67], the underlying hash function is considered as a component that can be replaced if it is found to be weak or when new more efficient or secure hash functions are developed. HMAC has been

chosen as the mandatory-to-implement MAC for IPSEC and is used in other Internet protocols such as Secure Socket Layer (SSL).

To obtain the MAC there should be minimum of three hash operations. We choose to investigate the implementation of HMAC using the full loop unrolled SHA-1 core. In the same manner, any architecture of SHA-1, or MD5 or other Internet specified hash algorithm could be used as the underlying hash function.

4.4.1 Design Description

In the HMAC-SHA-1 design, the SHA-1 core is used to obtain the hash operation. Similarly it is also possible to use the iterative SHA-1 or iterative or full loop unrolled MD5 as the cryptographic hash function for HMAC. The block diagram of the design is shown in Figure 39. The full loop unrolled core is a sub module in the "HMAC_SHA-1". The other main component of the design is the 16-state "HMAC_ state_ machine". The two counters are used for the same purposes explained under SHA-1 implementation.

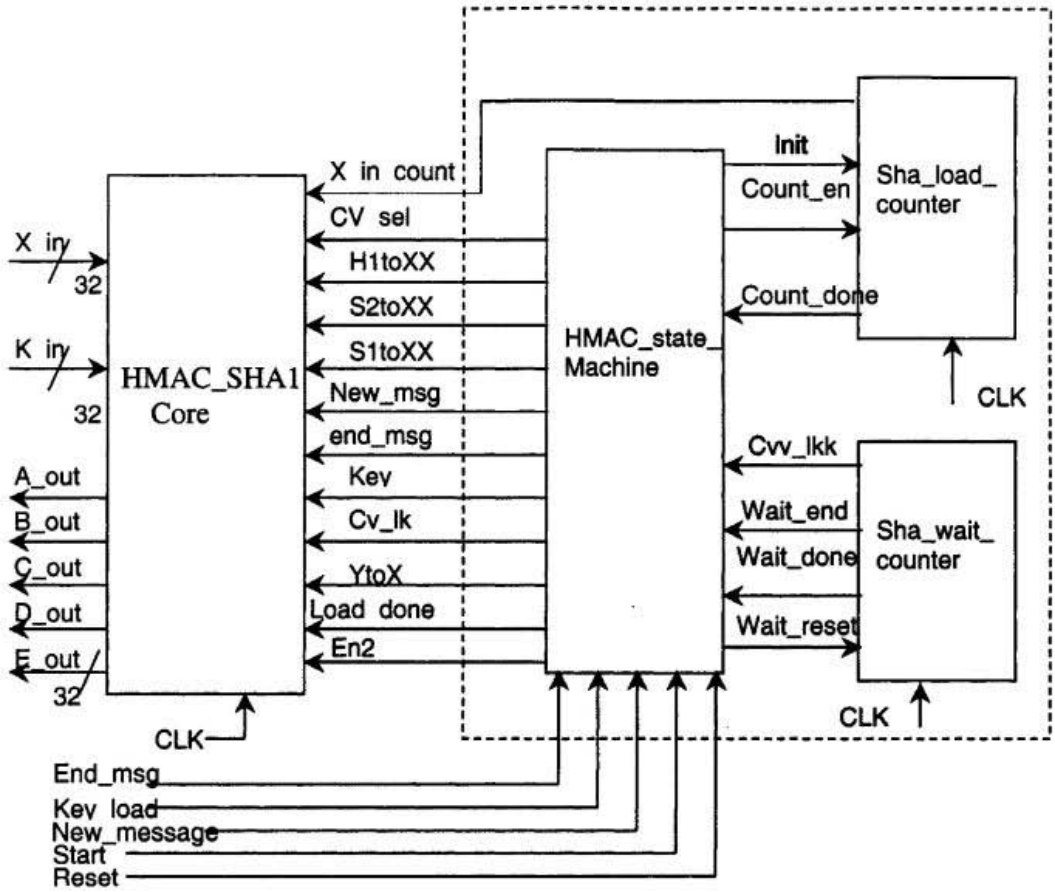


Figure 39. HMAC-SHA-1 Block Diagram

An important component in this design is the 16-state HMAC state machine for which the basic state diagram is given in Figure 40. The "Reset" signal initializes the state machine and counters. With the "Start" and the "Key_load" signals asserted, the particular key is loaded to the "HMAC-SHA-1". Using this key, the initial blocks for the two hash operations are prepared and stored in two arrays (S1 and S2) after which the first block is loaded and the computation is started.

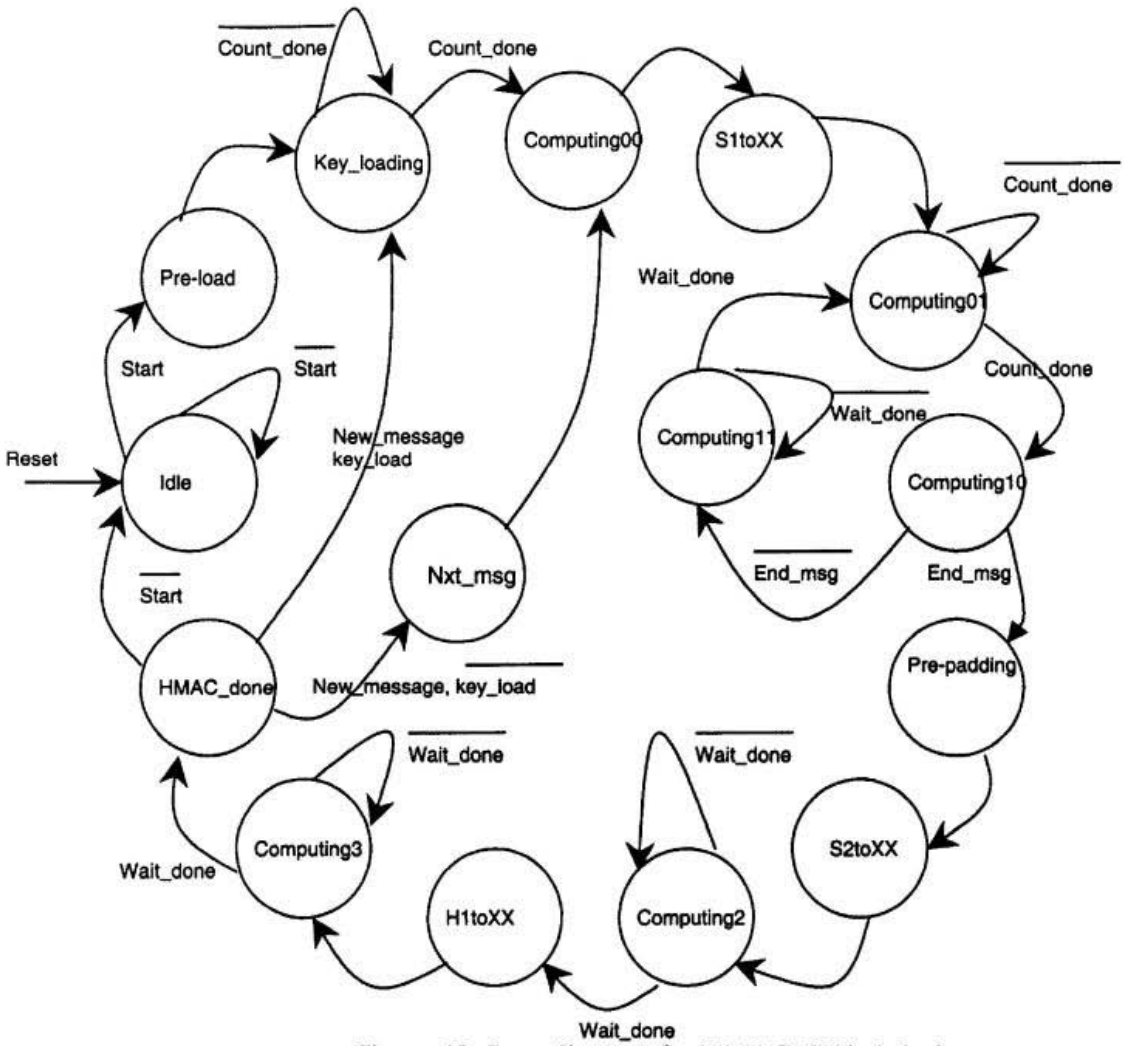


Figure 40. State diagram for HMAC-SHA-1 design

During the computation the next block is loaded. The signal "S1toXX" loads the next block to the core after the computation. This is continued until the full length of the message has been processed. When the end of the message is met ("End_msg"), the resultant hash value is padded so that the length is 512 bits. This is appended to the second initial value, which has already been created using the key. These two blocks are

hashed to create the MAC for the message. The "S2toXX" and "HtoXX" signals load the first and second blocks respectively, at the appropriate states. "SHA_load_counter" is used to load 16-word blocks to the core. "X_in_count" is the address of the block words for loading into and reading from the core. "SHA_wait_counter" counts the time required for a computation of one CV. The MAC will be presented as four 32-bit outputs, "A_out", "B_out", "C_out", "D_out" and "E_out". Once the HMAC for a message is done, if there is no key change, the computation can continue for the next message. Otherwise a new key can be loaded using "Key_load" signal. The MAC can be chosen as the complete HMAC result or a part of that depending on the requirement.

4.4.2 Simulation, Synthesis and Implementation Results.

The simulation, synthesis and implementation of HMAC-SHA-1 were performed using the same design flow applied for implementations of previous designs. The functional simulation result of the design is shown in Figure 41. The test case used for this simulation utilizes a 25-byte key and produces a 20-byte HMAC. The test vectors and the simulation details are as follows;

Test message: CD repeated 50 times.

Key: 0102030405060708090a0b0c0d0e0f10111213141516171819

The key and the message are loaded when the "New_message" and "Key_load" go high when the "reset" signal is low. At the end of the key load, "S1toXX" goes high and the

array "XX" takes the initial block value, which is the XOR of the extended key and "ipad". The first chaining variable is calculated using the initial value and the result can be seen in the registers "inter11", "inter22", "inter33" and "inter44". At the end of the computational time (64 clocks in this case) the signal "YYYtoXXX" goes high loading the message block into array "XX". At the end of calculation "S2toXX" becomes high and the XORed value of the extended key and "opad" is loaded to array "XX". At the end of this computation "H1toXX" signal becomes high and the padded hash value of the message is loaded to array "XX". As the "Start" is low, the HMAC value has appeared at the output.

The design was initially synthesized and implemented with medium effort and then it was synthesized several times with high effort and "period" timing constraint. The design interface of HMAC-SHA-1 is shown in Figure 42.

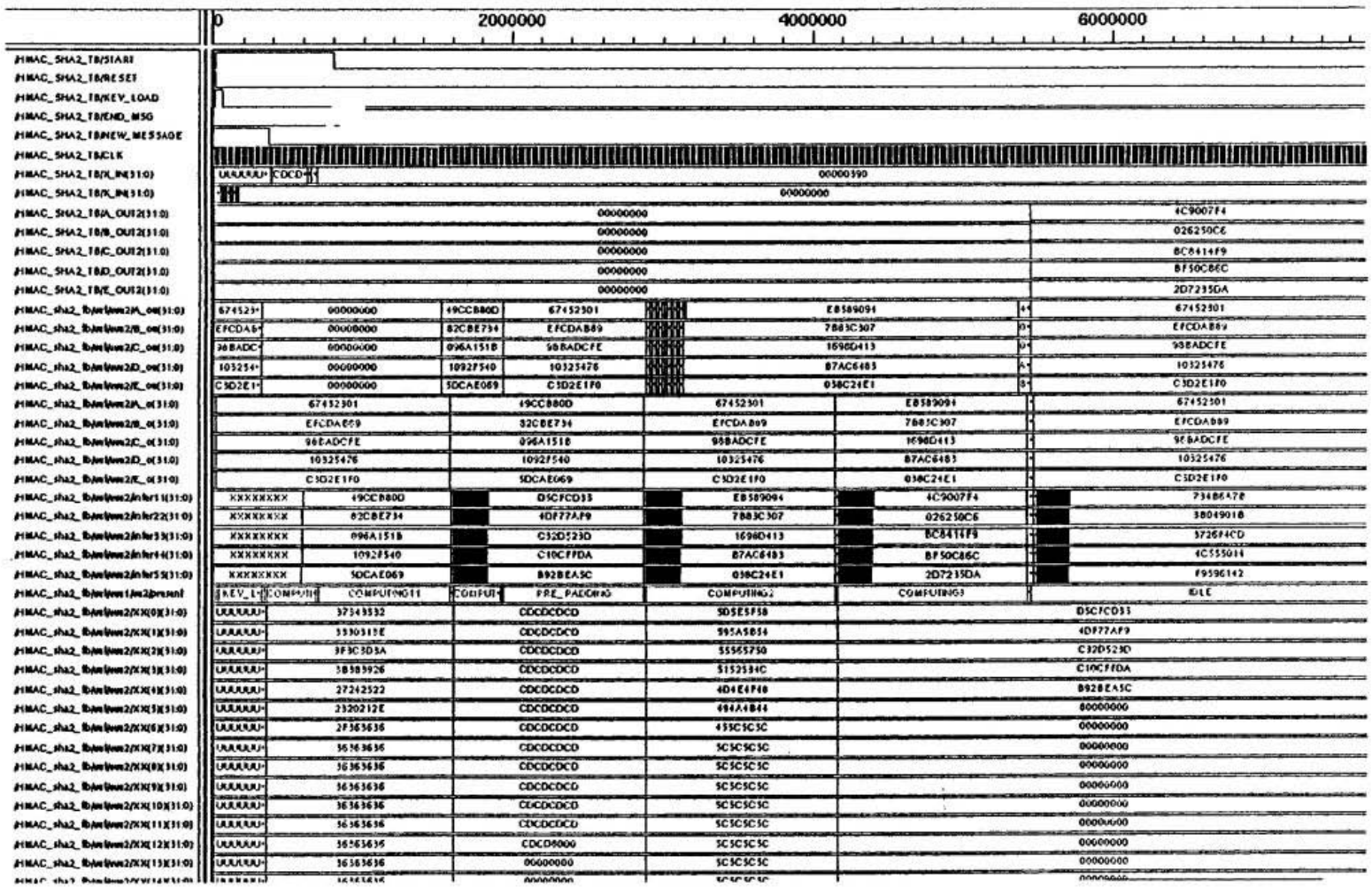


Figure 41. Functional simulation of HMAC-SHA-1

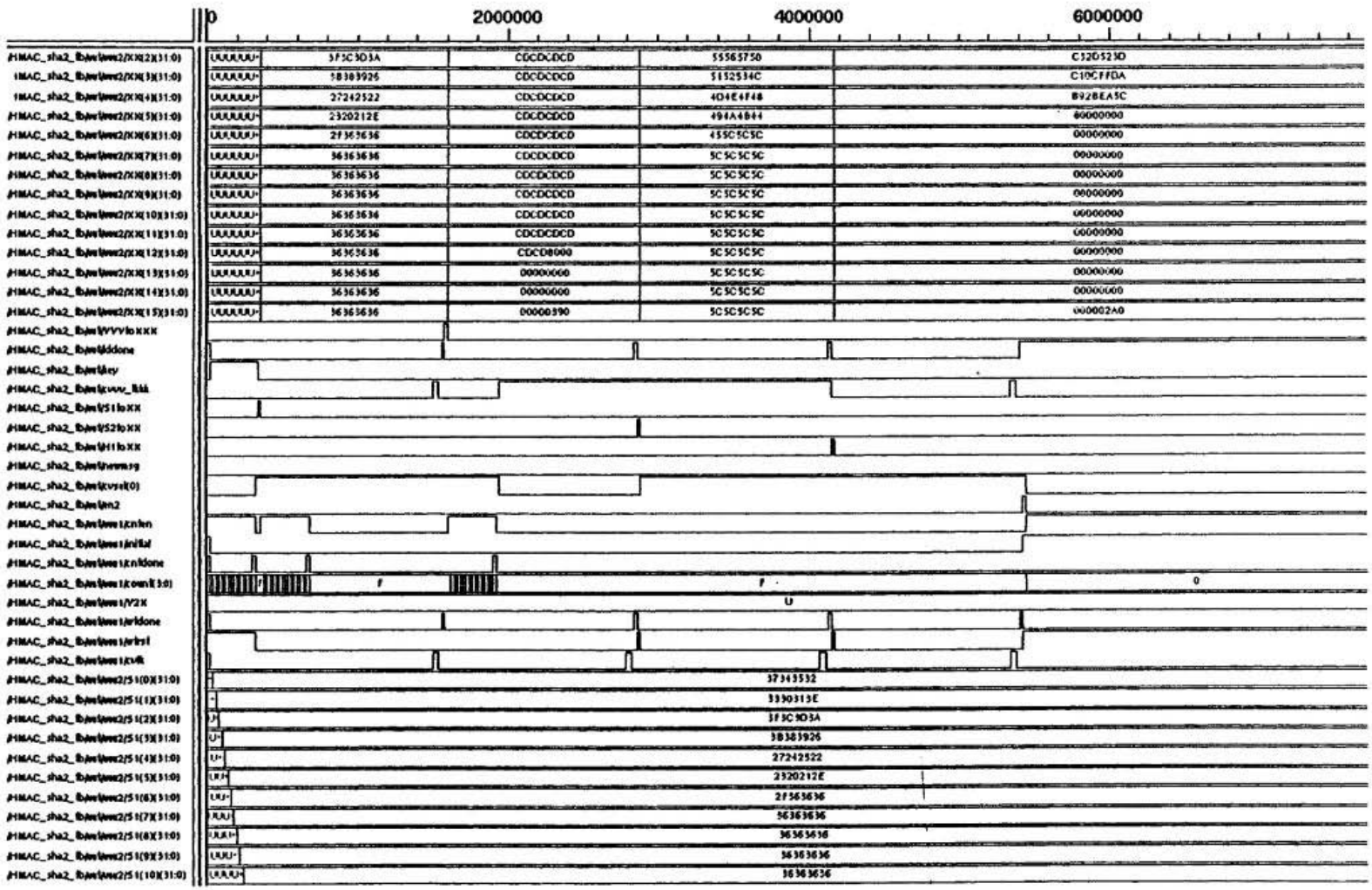


Figure 41. Functional simulation of HMAC-SHA-1 (contd)

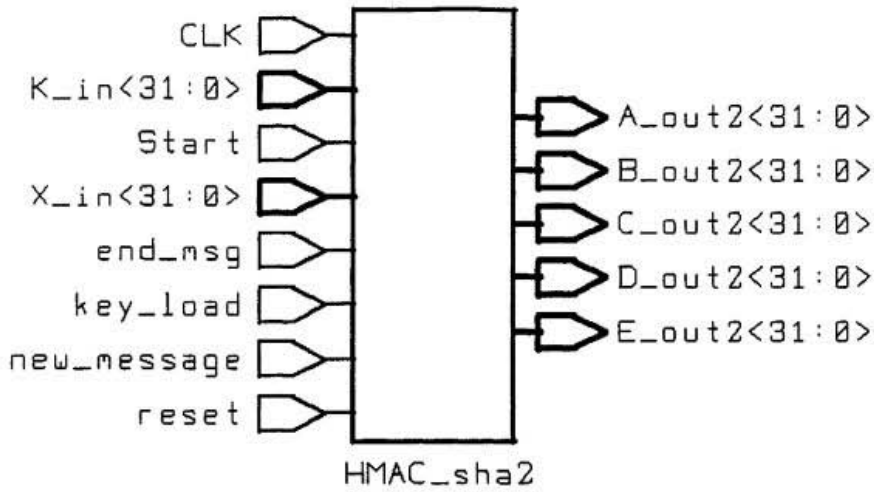


Figure 42. Interface of HMAC-SHA-1 design

The PAR results indicate that, in the cases where different efforts and constraints were used, the slice utilization is 7517 out of 12288 (61%) and IOB utilization is 229 out of 512 (44%). So the design is well fitted to a single device. According to Table 5 the effort level and period constraint has improved the performance by about 25%.

Table 5. Timing report summary of HMAC-SHA-1 design

	Medium Effort	High Effort	High effort & Period Constraint
Number of Slices (% of total)	61 %	61 %	61 %
Number of I/O Blocks	44 %	44 %	44 %
Logic Delay (% of total)	36.4%	45.1%	52 %
Routing Delay (% of total)	63.6%	52 %	48 %
Logic levels	1113	1235	1305
Total Delay (ns)	1212.9	1021.5	900.5

4.5 Performance Analysis of HMAC-SHA-1

The timing diagram for HMAC-SHA-1 operation for one single block message is given in Figure 43.

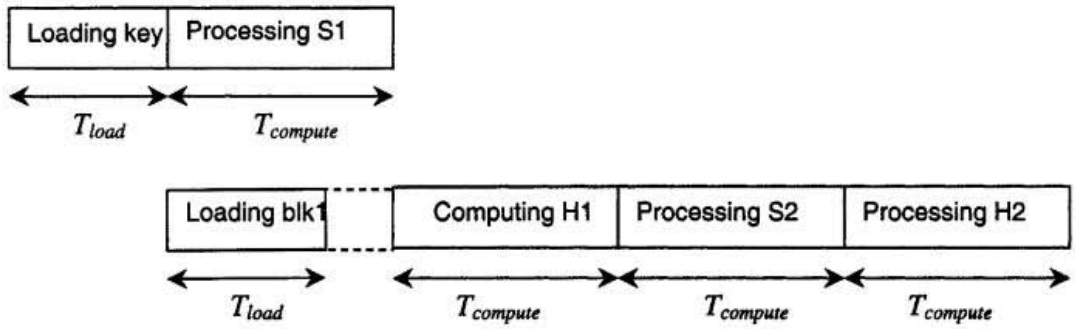


Figure 43. Timing diagram for HMAC-SHA-1 operations.

According to Table 5, though timing optimizing has increased the number of logic levels, it has not affected the slice utilization and routing delay has been reduced significantly. According to the area utilization, the design can be fitted to a single Virtex device. The minimum total delay was 900.5 ns. The throughput of this depends on the length of the message. As the length of the message increases the throughput gets closer to the throughput of in-built SHA-1. Hence if the length of the message is N_{blk} ,

$$\text{Throughput} = (N_{blk} \times 512)/((N_{blk} + 3) \times T_{compute}) \text{ bits/sec}$$

From Table 5, $T_{compute} = 900.5$ ns. Therefore,

$$\text{Throughput} = (N_{blk} \times 512) / ((N_{blk} + 3) \times 900.5 \times 10^{-9}).$$

For large messages the throughput would be almost as 568 Mbps, which is the throughput of the underlying SHA-1.

In the same way, the maximum throughput of HMAC with MD5 (full loop unrolled) as the underlying hash function can be estimated as $(N_{blk} \times 512) / ((N_{blk} + 3) \times 1054.12 \text{ ns}) = 485$ Mbps when $N_{blk} \rightarrow \infty$.

4.6 Conclusion

The five designs of the three algorithms have been successfully synthesized and implemented. Each design can be easily fitted into a single Virtex device. Some of the optimizing methods have been explored to improve the timing performance. The "Period" constraint has been effective for all the implementations. When a design comprises components, which can be generated by parameterizing the library components, the design can be well optimized using the features of the Xilinx tools. This is because the components are well mapped into the target device and hence the less amount of logic delay in the total delay of the critical path. The ability to reduce the routing delay has become a major reason for better performance of SHA-1 over MD5. Due to the reasons mentioned in section 4.3, the full loop unrolled design of SHA-1 could be optimized better than MD5. However the total critical path delay of the iterative core of SHA-1 was higher than that of MD5 due to larger number of logic levels in the iterative core of SHA-1 compared to MD5. Therefore in iterative designs, MD5 has a better timing than SHA-1.

The throughput values obtained for the three algorithms can meet most of the currently available IP bandwidths. Hence, FPGA implementation can be used as components in cryptographic accelerators for use in IPSEC. The results can be further

improved using more constraints, enabling delay based cleanup router passes and hardware floor planning. The use of latest FPGA devices with more resources and better speed performance will also provide better results. These three altogether are widely used in Internet security, which deals with a large range of message lengths. The size of the message has a significant impact on the performance of these algorithms. The next chapter discusses the performance of HMAC and CBC-MAC considering the characteristics of Internet traffic.

Chapter 5

Performance of MAC Algorithms for IPSEC

The cryptographic algorithms employed in Internet security must be able to handle packets which may vary in size over a large range. Most of the cryptographic hash algorithms process messages partitioned into blocks. Due to this fact the messages have to be prepared by padding the required amount of zero bits to get an integer number of blocks. The length of the message is also appended at the end. This process becomes a considerable overhead when the short messages are more dominant in the message stream. Hence the size of the message has a significant impact on the performance of such algorithms.

The statistical properties of Internet traffic are complex and the amount of data to be studied is very large. The understanding of Internet traffic is useful to study the performance of authentication algorithms. In this chapter we will analyze the performance of HMAC and CBC-MAC in the context of the traffic characteristics of the Internet.

5.1 Previous Studies of Internet Traffic

Various attempts have been made over the years to study the nature of the wide area Internet traffic. Many of the studies have used Internet traffic trace data as input to evaluate specific protocol performance issues but have not concentrated on characterizing the underlying traffic contained in the traces. Among them Feldmann et al. [67] have

performed a detailed characterization of Hyper Text Transfer Protocol (HTTP) response traffic since the World Wide Web has a dominant influence in network dynamics. Figure 44 highlights the basic traffic characteristics of incoming and outgoing packets obtained by Feldmann et al. for a one-week trace. According to this study, half of the outgoing traffic consists of 40-byte Transport Control Protocol (TCP) acknowledgment packets, contributing to a low average packet size of 123 bytes. The remaining short packets are mainly HTTP requests. One-fifth of the incoming packets stem from 40-byte TCP acknowledgements. According to this study, more than 60% of the incoming packets have 552, 576 or 1500 bytes, that correspond to common maximum transfer unit sizes on the Internet [67].

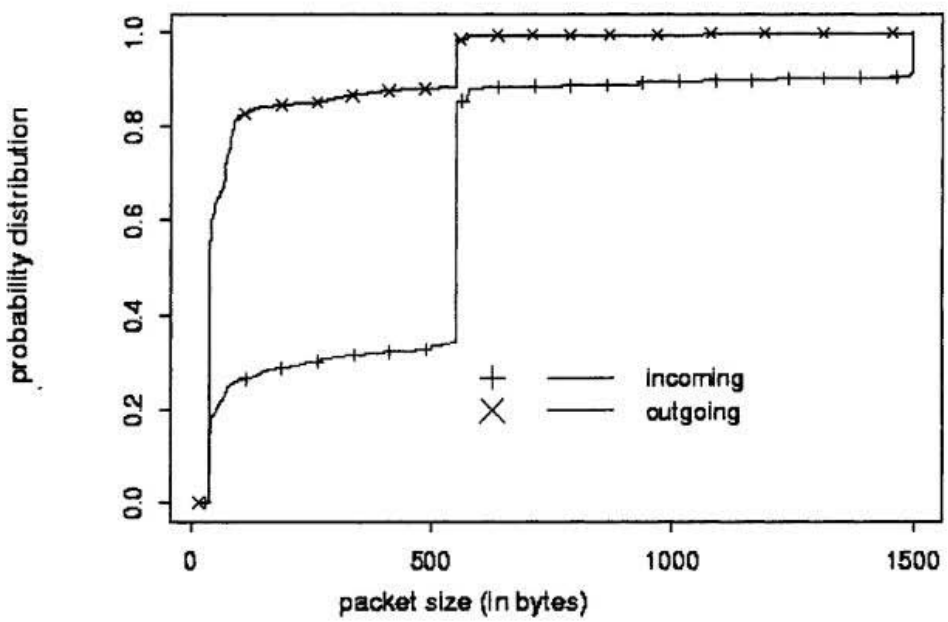


Figure 44. Cumulative distribution of packet sizes [67].

Several important observations have been made by the analyses of actual traffic carried in the Internet backbone. The Corporative Association for International Data Analysis (CAIDA) has performed studies on backbone traffic characteristics at a site inside a major Internet traffic exchange point over a period of 10 months [68]. The traces for the study were collected from an OC-3 ATM link using an optical splitter. The distribution of IP packet sizes is shown in Figure 45.

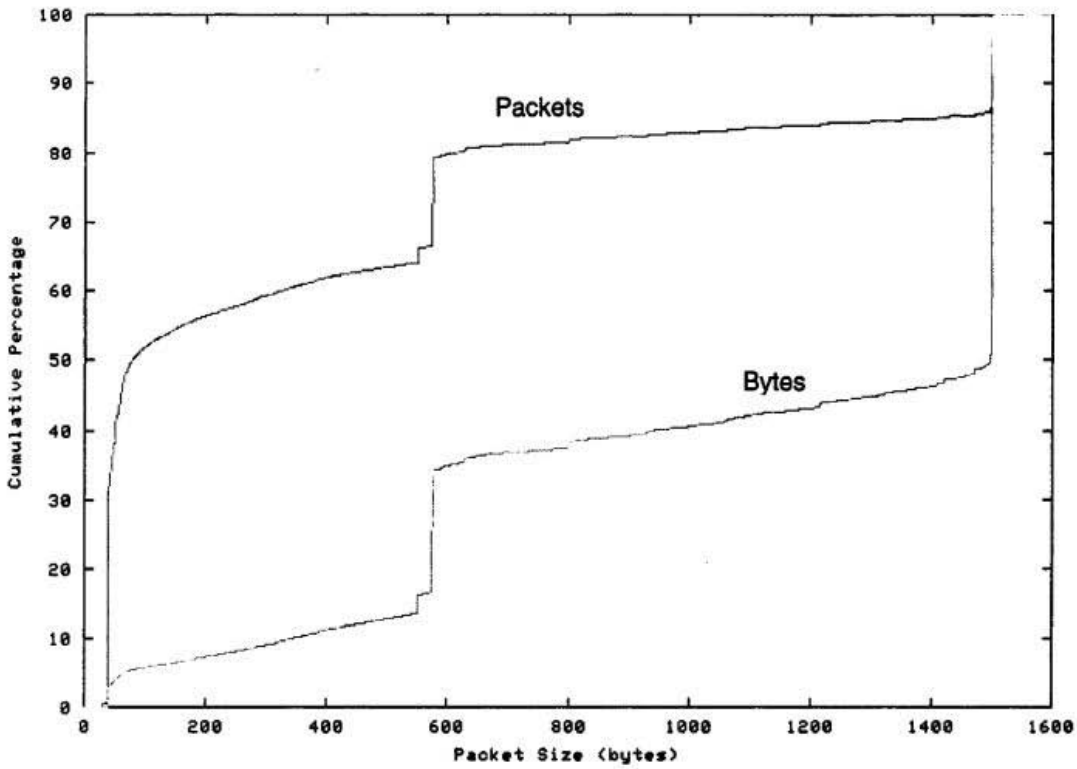


Figure 45. Cumulative distribution of IP packet sizes[69].

According to this study, TCP has contributed to about 85% of the traffic in the traces. A large portion of TCP traffic has been generated by bulk transfer applications such as HTTP and FTP. The packet size has been categorized into three groups namely

40 byte packets (minimum packet size for TCP), which carry TCP acknowledgments but no payload, 1500 byte packets (maximum Ethernet payload size) and 552 and 576 byte packets, from TCP.

A rule of thumb employed in some of the IP traffic analysis has three packet sizes each having equal probability of occurrence. They are 40 bytes, 256 bytes and 1500 bytes, each having a probability of occurrence of one third [70].

5.2 IP Packet Size Models

Studying the behavior of IP packet size of the Internet messages allow the estimation of actual performance of authentication functions such as HMAC-SHA-1 or CBC-MAC. The average block size can be derived by analyzing the statistical observations. For analytical purposes the probability density function (PDF) of the IP packet size will be approximated as one of the following four cases:

Case (i)

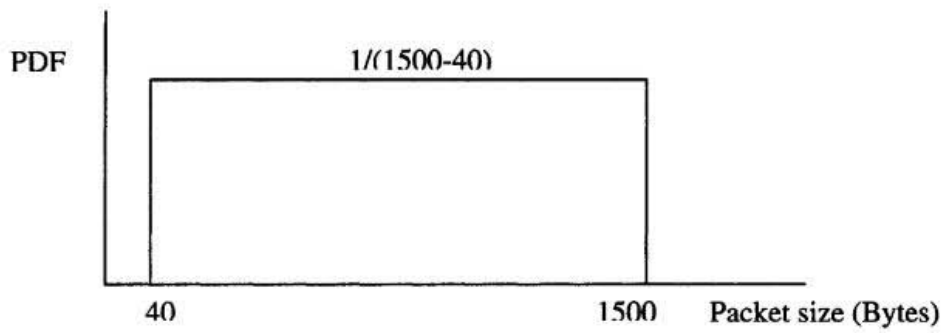


Figure 46. Uniform PDF

Case (ii)

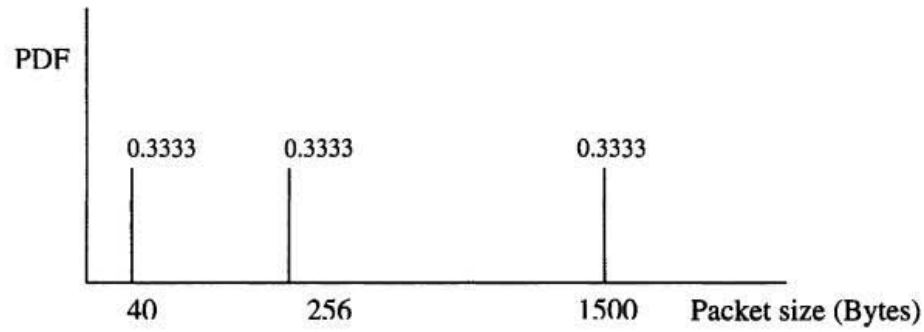


Figure 47. Rule-of- thumb of PDF

Case (iii)

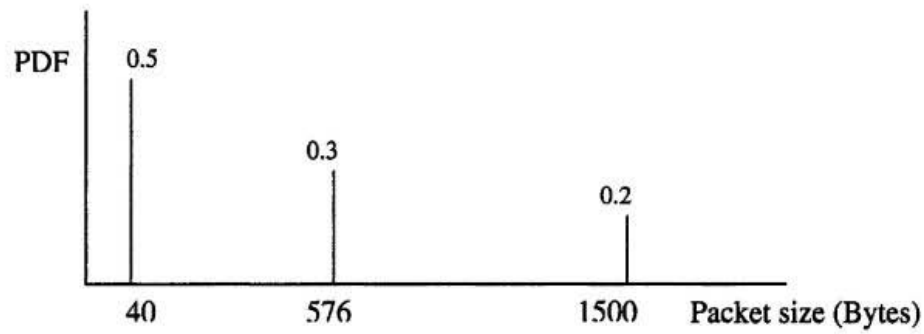


Figure 48. Discrete PDF with 3 impulses

Case (iv)

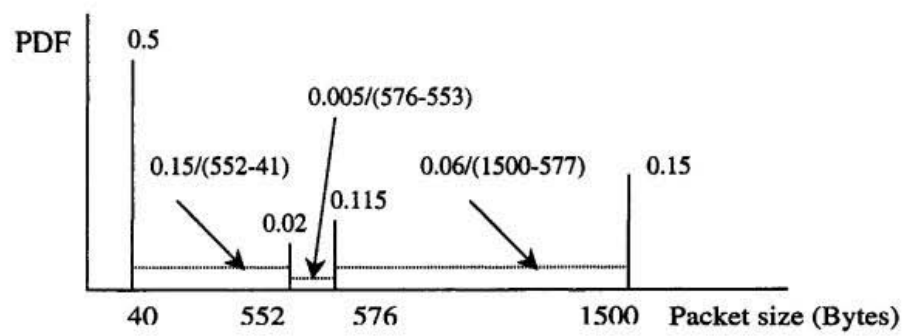


Figure 49. Discrete and Uniform PDF

Each case has a different degree of accuracy in modeling Internet traffic. Case (i) is the straightforward assumption of uniformly distributed packet sizes; Case (ii) is a typical rule-of-thumb; Case (iii) is a better rule-of-thumb; Case (iv) follows most closely Figure 45.

The average number of blocks per packet (Γ) can be obtained as follows. If the number of blocks of the message of packet size i bytes is $Nblk(i)$ and the probability of the message has i bytes is $p(i)$,

$$\Gamma = \sum_{i=40}^{1500} p(i) \times Nblk(i) \tag{5.1}$$

5.3 Performance of MACs in Internet

HMAC-SHA-1 has been recommended for message authentication in several network security protocols. The key reasons behind this are the provable security, free availability, flexibility of changing the hash function and reasonable speed, among others. The MAC based on the block ciphers such as CBC-MAC-DES [34] was generally considered as slow due to the complexity of the encryption process. As well, since DES has already been broken it is not recommended any more. However, after selecting the AES encryption algorithm, this situation merits reevaluation as Rijndael [49] shows good performance in both hardware and software and it has better security features than DES. CBC-MAC is likely to be standardized as an AES mode of operation. For HMAC one block has a size of 512 and for CBC-MAC analysis, we assume a block size of 128 bits (as is the case for AES).

The CBC-MAC comes in different versions varying in details such as padding, length variability and key search strengthening [71]. The general way of padding for CBC-MAC is by considering the final input block as a partial block of data, left justified with zeroes appended to form a full block [34]. Let us take the number of blocks of the message of packet size i bytes for HMAC and CBC-MAC using AES (CBC-MAC-AES) are $Nblk_{MAC}$ and $Nblk_{AES}$, respectively. According to the padding method described in Section 2.2.5, $Nblk_{MAC}$ can be obtained as follows:

$$Nblk_{MAC} = \begin{cases} \lceil (i\times 8)/512 \rceil + 1 & \text{if} \quad (i\times 8) \bmod 512 = 0 \quad \text{or} \\ & \quad (i\times 8) \bmod 512 \geq 448 \\ \lceil (i\times 8)/512 \rceil & \text{if} \quad (i\times 8) \bmod 512 < 448 \end{cases}$$

Assuming the general way of padding described above has been adopted for CBC-MAC-AES,

$$Nblk_{AES} = \lceil (i \times 8)/128 \rceil.$$

5.3.1. Average Number of Blocks per Packet

The average number of blocks per packet of the IP traffic can be estimated according to the four cases mentioned earlier.

Case (i):

In this case the PDF of the packet size is assumed to be uniformly distributed. From Figure 46, the average number of blocks for HMAC can be estimated as follows. From equation (5.1),

$$\Gamma = \sum_{i=40}^{1500} [(0.5/1460) \cdot Nblk_{MAC}(i)]$$

$$= 6.34 .$$

Therefore the average number of HMAC blocks for this model is 6.34.

Similarly the average number of blocks for CBC-MAC can be estimated as follows. For 128 bit block size, $\sum_{i=40}^{1500} [Nblk_{AES}(i)] = 71086$. Hence from equation (5.1), the average number of CBC-MAC blocks for a uniformly distributed packet size, is Γ = 24.34.

Case (ii):

This is a rule of thumb used in some performance analyses related to IP traffic. From Figure 47, the average number of blocks for HMAC can be estimated based on:

$$Nblk_{MAC}(40) = 1$$

$$Nblk_{MAC}(256) = 5$$

$$Nblk_{MAC}(1500) = 24.$$

Therefore, the average number of HMAC blocks for the model used as a rule of thumb, is 10. The average number of CBC-MAC blocks also can be calculated in the same manner:

$$Nblk_{AES}(40) = 3$$

$$Nblk_{AES}(256) = 17$$

$$Nblk_{AES}(1500) = 94$$

Hence, the average number of CMC-MAC blocks for this model is 38.

Case (iii):

This model has a PDF with three impulses as shown in Figure 48. The average number of blocks for HMAC can be calculated using:

$$Nblk_{MAC}(40) = 1$$

$$Nblk_{MAC}(576) = 10$$

$$Nblk_{MAC}(1500) = 24$$

Therefore, using (5.1) the average number of HMAC blocks when the discrete PDF with 3 impulses model is used for the IP packet size, is 8.3. In the same manner the average number of blocks for CBC-MAC can be estimated by noting:

$$Nblk_{AES}(40) = 3$$

$$Nblk_{AES}(576) = 37$$

$$Nblk_{AES}(1500) = 94$$

Hence, the average number of CBC-MAC blocks for a packet size model with discrete PDF shown in Figure 48, is 31.4.

Case (iv):

This model closely follows the cumulative distribution given in Figure 45. From Figure 49 the average number of blocks for HMAC can be estimated using:

$$Nblk_{MAC}(40) = 1$$

$$Nblk_{MAC}(552) = 9$$

$$Nblk_{MAC}(576) = 10$$

$$Nblk_{MAC}(1500) = 24$$

From equation (5.1), the average number of HMAC blocks can be found as follows:

$$\Gamma = (0.5)(1) + \sum_{i=41}^{551}[(0.15/(511)\cdot Nblk_{MAC}(i)] + (0.02)(9) + \sum_{i=553}^{575}[(0.005/(23))\cdot Nblk_{MAC}(i)]$$

$$+ (0.115)(10) + \sum_{i=577}^{1499}(0.06/(923))\cdot Nblk_{MAC}(i) + (0.15)(24).$$

Where the summations of the number of blocks can be easily found as shown below:

$$\sum_{i=41}^{551} Nblk_{MAC}(i) = 2687\ ,\quad \sum_{i=553}^{575} Nblk_{MAC}(i) = 215\ ,\quad \sum_{i=577}^{1499} Nblk_{MAC}(i) = 15558\ .$$

Therefore, the average number of HMAC blocks with this model, which closely follows the actual packet distribution is used for the IP packet size, is $\Gamma = 7.28$.

In the same manner the average number of blocks for CBC-MAC can be calculated using:

$$Nblk_{AES}(40) = 3$$

$$Nblk_{AES}(552) = 35$$

$$Nblk_{AES}(576) = 37$$

$$Nblk_{AES}(1500) = 94$$

From Figure 46 and equation (5.1) the average number of CBC-MAC blocks is given by

$$\Gamma = (0.5)(3) + \sum_{41}^{551}[(0.15/(511))\cdot Nblk_{AES}(i)] + (0.02)(35) + \sum_{553}^{575}[(0.005/(23))\cdot Nblk_{AES}(i)].$$

Hence, the average number of CBC-MAC blocks for the model, which closely follows the actual packet distribution, is 27.51.

5.3.2 Performance in Hardware

The performance of the FPGA implementations of HMAC-SHA-1 (using full loop unrolled results of SHA-1) and CBC-MAC-AES is compared in the four IP traffic cases previously mentioned. The FPGA implementation results of Rijndael published by NIST have been used for this analysis. An FPGA implementation of Rijndael has been carried out using Xilinx Virtex XCV1000BG560-4 device by A.J Elbirt et al. [51] for the evaluation of AES finalist algorithms. The clock frequency that has been obtained by them for the loop unrolled architecture in feedback mode is 14.1 MHz. One block has taken 6 clock cycles and thus a throughput of 300.1 Mbps has been obtained. The delay for one block encryption is 426.5 ns. However, for comparison purposes the speed grade of the Virtex device has to be taken as –6 as with our implementations. The change of speed grade –4 to –6 gives approximately 28% of speed enhancement [60]. Hence, the delay for block encryption would approximately be 307.08 ns.

The average time for calculating MAC using HMAC and CBC-MAC-AES can be determined by the following relationships:

Average HMAC calculation time

$$= (3 + (\Gamma)) \times [\text{time for a hash}] \qquad (5.2)$$

Average CBC-MAC calculation time

$$= (\Gamma) \times [\text{time for a block encryption}] \qquad (5.3)$$

The calculated values for HMAC and CBC-MAC-AES using equations (5.2) and (5.3) are given in Table 6.

Table 6. Times for HMAC-SHA-1 and CBC-MAC-AES on FPGA for general IP traffic

	Case (i)		Case (ii)		Case (iii)		Case (iv)	
	Average packet size (in blocks)	Average packet time(μs)	Average packet size (in blocks)	Average packet time(μs)	Average packet size (in blocks)	Average packet time(μs)	Average packet size (in blocks)	Average packet time(μs)
HMAC-SHA-1	6.34	8.21	10	11.71	8.3	10.18	7.28	9.25
CBC-MAC-AES	24.34	7.48	38	11.67	31.4	9.64	27.51	8.45

5.3.3 Performance in Software

Both HMAC-SHA-1 and CBC-MAC-AES were run on a 927 MHz Pentium II machine. The C code which has been used for assessing the speed performance of Rijndael by NIST [72] was used in CBC mode for CBC-MAC-AES. The plaintext of 128-bit was encrypted 1,000,000 times in CBC mode using a 128-bit key. The average time for encrypting a 128-bit block was 1.14 μs. The C code of SHA-1 published by DI management [73] according to NIST specifications was used to determine the software speed of hashing. The average time taken for creating a hash value for a 512-bit block was 3.001 μs.

Table 7. Times for HMAC-SHA-1 and CBC-MAC-AES on software for general IP traffic

	Case (i)		Case (ii)		Case (iii)		Case (iv)	
	Average packet size (in blocks)	Average packet time(μs)	Average packet size (in blocks)	Average packet time(μs)	Average packet size (in blocks)	Average packet time(μs)	Average packet size (in blocks)	Average packet time(μs)
HMAC-SHA-1	6.34	28.03	10	39.01	8.3	33.91	7.28	30.84
CBC-MAC-AES	24.34	27.75	38	43.32	31.4	35.80	27.51	31.36

5.4 Conclusion

According to Table 6 the average times of MAC calculations of the FPGA implementations of HMAC and CBC-MAC-AES do not differ significantly. Particularly, they are almost the same as the size of the packets becomes larger.

The software results given in Table 7 show that the timing performances of HMAC-SHA-1 and CBC-MAC-AES do not have a significant difference. As observed in FPGA implementations HMAC offers better timing performance as the packets become larger. In general, the size of the packet has a considerable impact on the performance of cryptographic hash algorithms as the padding has to be carried out all the time even if the message has a length of multiples of 512 bits. This becomes a large overhead for small

packets especially in case of HMAC algorithm. Most of today's cryptographic algorithms follow a sequential structure, which is difficult to pipeline. Many of them cannot meet today's speed requirements. Hence the new approaches of hash algorithms, which are more efficient in speed, have to be explored. The next chapter discusses one of the new approaches in message authentication.

Chapter 6

A New Approach: Universal Message Authentication Code

For many applications sufficient speed has already been obtained from algorithms such as HMAC-SHA-1 [67] or CBC-MAC of a block cipher [33]. However for the most speed demanding applications some alternative methods have to be identified. The well known technique for message authentication using universal hash functions seems to be very promising as it provides schemes that are both efficient and provably secure under reasonable assumptions [74]. During the last few years, progress has been made both in theory and practice of universal hash functions. For example, Universal Message Authentication Code (UMAC), a non-cryptographic hash algorithm has been developed. This does not need to have any cryptographic hardness property but some combinatorial properties that can be proved [46]. Halevi and Krawczyk have developed a very fast scheme, multi-linear modular hashing (MMH) which makes optimal use of multiply and accumulate instructions of the Pentium MMX processor [45] which provide small-scale single instruction multiple data (SIMD) parallelism in its instruction set.

Recently, Back et al. have further improved the performance on high-end processors with Universal Message Authentication Code (UMAC) construction, which is faster than MMH on processors with fast multiplication [46]. The authors of UMAC report a software performance of 5.6 G bits/sec on a 350 MHz Pentium II. In this chapter the structure and the complexity of the UMAC algorithm, in terms of hardware implementation, are discussed.

6.1 UMAC Construction

UMAC is a parameterized algorithm. Various low level choices have not been fixed. The values for these parameters are chosen before the authentication generated by UMAC becomes well defined. Hence instead of making generic compromises, an application that uses UMAC can choose the parameters which best suit its requirements or implementation environment. There are two sets of parameters chosen as UMAC-16 and UMAC-32. UMAC-16 is designed to exploit small-scale SIMD parallelism found in modern processors and UMAC-32 is designed to do well on processors with good 32-bit and 64-bit support. There are six basic parameters as given below.

Word length: The size of word in bytes can be either 2 or 4 bytes. UMAC-32 uses 4 bytes.

UMAC output length: Specifies the length of the authentication tag in bytes. It can be any value between 1 and 32 bytes. The default value is 8 bytes.

Block length: Specifies the message block length in bytes, which can be any value between 32 and 2^{28} bytes. The default value is 1024 bytes.

UMAC key length: Specifies the user supplied key length in bytes. This can be either 16 or 32 bytes. The default value is 16 bytes.

Endian Favorite: Specifies which endian orientation is used in reading data.

Operations sign: Specifies whether string operations are signed or unsigned.

Moreover UMAC offers a tradeoff between forging probability and speed, which is not common in the construction of other authentication algorithms. UMAC also enjoys better analytical security properties than the existing authentication algorithms.

The overview of UMAC operation is as follows. The message to be authenticated is first applied to the universal hash function (UHASH) resulting in a string, which is much shorter than the original message. The UHASH has three layers, which use different sizes of keys. The keys have to be derived from the shared key through a pseudo random function (PRF). The same PRF is used for the tag generation. The PRF is applied to a nonce and the authentication tag is the logical xor of the two outputs of the hash function and the pseudorandom function. The nonce is a parameter such as a counter that varies with time. It is widely applied in key management protocols to prevent message replay and other types of attacks. The authentication tag is generated as

$$\text{tag} = f(\text{nonce}) \text{ XOR } h(\text{message})$$

where f is the pseudorandom function and h is the universal hash function shared by the sender and the receiver. The sender does not reuse the nonce under the same MAC key. Typically the nonce would be incremented using a counter with each message. UMAC defines the creation of authentication tag using the message, shared key and nonce. The receiver can recompute the tag using the received message and nonce to see the authenticity. A shared key is used to key the PRF, f, to generate all of the pseudorandom bits required by the layers of the universal hash function. Therefore the f is used for generating both the tag and all of the pseudorandom bits used in the algorithm [75].

6.1.1 UMAC Key Derivation

The user-supplied key is expanded into the internal keys using the key setup process. The total keys needed for all iterations are found by repeating this process until the required key lengths are achieved. Key setup can be carried out using a block cipher or some cryptographic hash function. Example block ciphers are Rijndael, DES, RC6 and MARS. Examples of hash functions are HMAC-SHA-1, SHA-1, MD5 and RIPEMD-160 [46]. The Internet draft of UMAC [75] describes the key derivation using AES block cipher (Rijndael) in output feedback mode to produce pseudorandom bits needed within universal hashing. Depending on the size of the user supplied key the 128-bit and 256-bit block length variants of AES cipher are used. An index is used so that using the same key different pseudorandom outputs can be generated with different indices.

6.1.2 Tag Generation

The output of the universal hash function is XORed with a pseudorandom string to produce the tag. The pseudorandom string is generated by applying AES block cipher to a nonce. The nonce can be 1 to 16 bytes of length but all nonces in a single session should be of equal length. AES may provide more or fewer bits per invocation than are needed for the tag generation. Then one part of the AES output could be used to generate one tag and the other part could be used for the tag of the next message. This optimization is effective only when the nonces are sequential. The low bits of the nonce are used as an index into the AES output, which is generated using higher bits of the

nonce, which are not used for indexing. If the UMAC output is larger than 16, then two AES invocations are required to produce sufficient number of bits [75].

6.1.3 Universal Hash Function (UHASH)

UHASH is the keyed hash function, which takes a message of an arbitrary length and produces a string of fixed length (e.g. 8 bytes). This has been constructed in three layers.

1. Non-linear hash (NH) compresses input messages into strings, which are typically many times smaller than the input messages.
2. The compressed message is hashed with an optimized "polynomial hash function" into a fixed length of 16-byte string.
3. The 16-byte string is hashed using "inner product hash" into a string of length 2 or 4 bytes, which is repeated to the appropriate tag length.

Figure 50 shows the generic interface of UMAC for 32-bit operation.

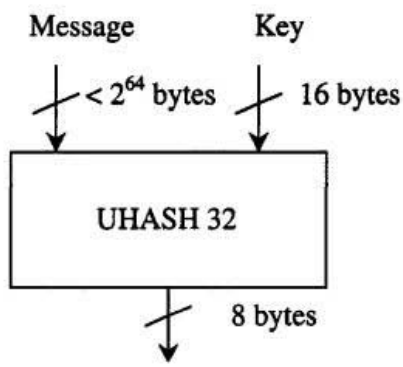


Figure 50. Interface of UHASH

Figure 51 shows the structure of UMAC diagrammatically. This diagram represents one of the applications of UMAC in 32-bit architecture. Here the block length on which the hash function initially operates is 1024 bytes.

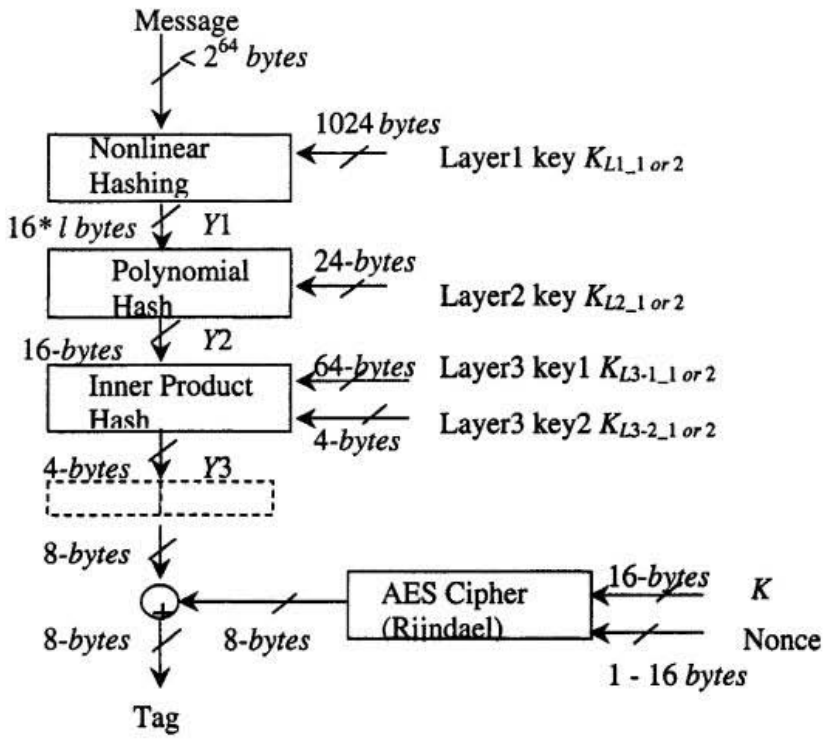


Figure 51. General structure of UMAC

In the figure, l is the number of blocks in the message. In this case the three layers have to be run twice with different set of keys to get the required 8 bytes output (shown in dotted lines). For the tag generation, half of the AES output bits could be used and the second half of the same could be used for the tag of the next message. This optimization is effective only when nonces are sequential.

These three layers are repeated with a modified key until the required output length is obtained. Since this repetition is independent, each word of the final output can

be computed independently. Hence computing a prefix of the tag can be done significantly faster than computing the whole tag. If the message being hashed is no longer than 1024 bytes (for the case shown in Figure 51), then Layer 2 hashing is skipped as an optimization. To reduce memory requirements the first and the third layers reuse most of their key material between iterations. The values used for the UMAC parameters in following description are related to UMAC-32 construction.

Layer 1: Nonlinear Hashing (NH)

This is designed to be fast on modern processors. NH hashes an input string M using a key K and performing a sequence of arithmetic operations. There are two versions, NH-16 and NH-32 depending on the *word length* parameter of UMAC. Consider a message $M = M_0, M_1,..M_l$ where $M_i \in \{0,..2^{32}\text{-}1\}$. In order to limit the length of key required in the first layer, the message is broken up into chunks no longer than the key length of Layer 1. Each chunk is hashed with a key of the same length. NH-32 is constructed using a sequence of $n \geq l$ 32-bit sub keys derived from the shared key K. The Layer 1 keys are represented as $K_{L1}= (K_0, ..K_n)$.

$$\mathrm{NH}_{KL}(M) = \sum_{i=1}^{l/2} \{[(M_{2i-1} + K_{2i-1}) \bmod 2^{32} \times (M_{2i} + K_{2i}) \bmod 2^{32}]\} \bmod 2^{64}$$

Here the operations are 32-bit and 64-bit addition and 32-bit multiplication. These are modular operations on 2^{32} and 2^{64}. Figure 52 shows the NH-32 construction for 256-bit message string. All the string sizes are given in bytes.

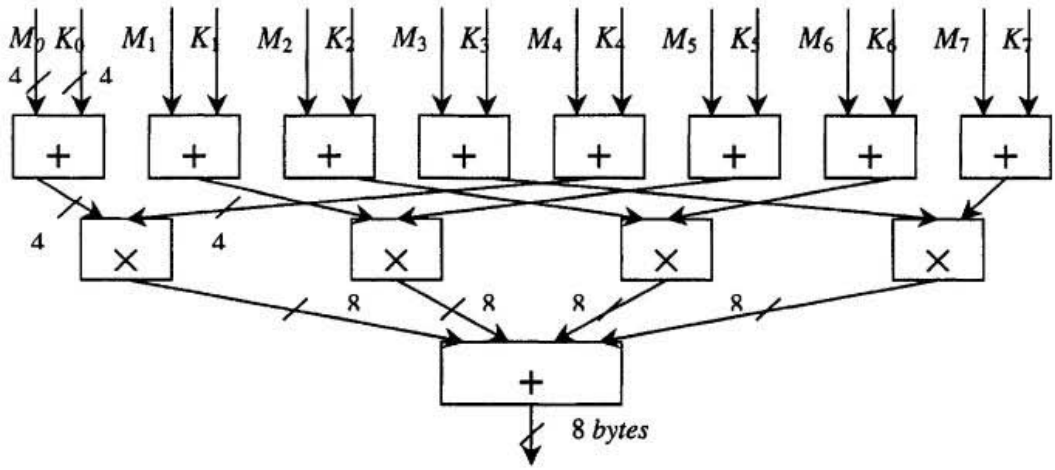


Figure 52. NH-32 construction for 256-bit message string

The input message is broken into chunks of *Block length* bytes (e.g. 1024 bytes). The last block is zero padded to an appropriate length. Then each chunk is hashed with NH and the outputs from all the invocations are annotated with 8-byte length information of the message block. This produces the Layer 1 output ($Y1$) as shown in Figure 53.

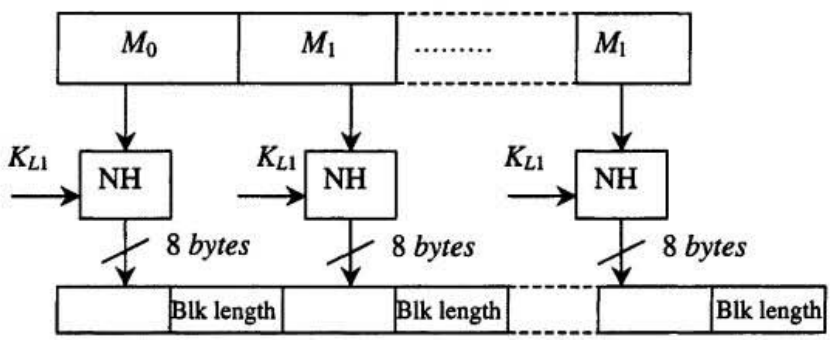


Figure 53. UHASH Layer 1

With respect to hardware implementation, the major operations involved in Layer 1 are the 32-bit addition in modulo 2^{32}, 64-bit addition in modulo 2^{64} and 32-bit multiplication in modulo 2^{64}. The complexity is discussed in Section 6.2.

Layer 2: Polynomial hashing.

The overview of Layer 2 is shown in Figure 54.

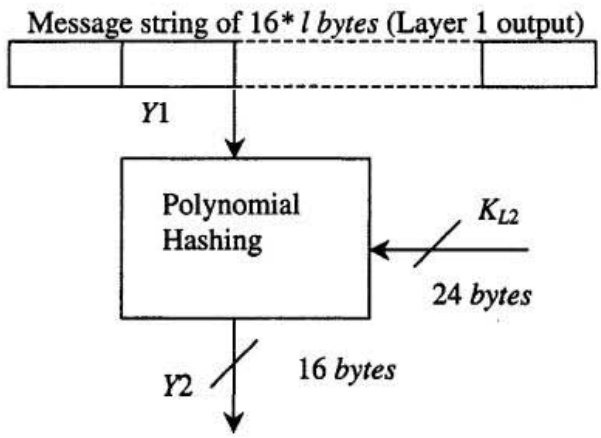


Figure 54. UHASH Layer 2

The output of Layer 1 hashing is still has a considerable length (16 × (number of message blocks) bytes). Polynomial hashing is used to reduce this to a fixed length of 16 bytes. In this hashing the input string of bits is treated as a sequence of coefficients of a polynomial and the hash key is the point at which the polynomial is evaluated. The security assured by polynomial hashing degrades linearly in the length of the message being hashed. The collision probability of two messages of n words in polynomial hashing with a prime modulus p is no more than n/p [75]. There should be some way to ensure the collision probability does not increase beyond a certain limit. By dynamically

increasing the prime modulus used in polynomial hashing the collision probability bound is achieved.

As Layer 1 produces quite long strings and the security guarantee of polynomial hashing degrades linearly, some scheme is required to allow long strings while ensuring that the collision probability never grows beyond a certain preset value. However, polynomial hashing under a small prime modulus is faster than hashing under a larger one. Hence the prime modulus is increased according to a preset value so that short messages are hashed faster still accommodating long ones [75]. In UMAC-32 if the length of the message is $\leq 2^{17}$ bytes, then only the prime number $(2^{64} - 59)$ is used for hashing. Otherwise the first 2^{17} bytes are hashed under $(2^{64} - 59)$ and remainder under $(2^{128} - 159)$. A 24-byte Layer 2 key (K_{L2}) is utilized to obtain two keys of 8 bytes and 16 bytes, which are required, in polynomial hashing under prime numbers $(2^{64} - 59)$ and $(2^{128} - 159)$ respectively. For this purpose two 8-byte and 16-byte masking values are used.

$$k_{64} = (K_{L2}[1..8]) \text{ AND (8-byte Mask)}$$

$$k_{128} = (K_{L2}[9..24]) \text{ AND (16-byte Mask)}$$

The polynomial hashing algorithm takes a string of bits of length divisible by 4 bytes, the prime number (p), the integer value of key k (which can be either k_{64} or k_{128}) and an integer (*maxrange*) to adjust the word strings so that their values are always less than the prime number and produces an integer in the range of $[0.... p-1]$. Any word larger than *maxrange* is split into two words and guaranteed to be in the range. The *maxrange* would be either 59 or 159 depending on whether the prime number is $(2^{64} -$

59) or (2^{128} –159), respectively. The input word string size can be either 8 bytes or 16 bytes and accordingly the prime numbers are chosen.

If the integer values of each 8-byte (or 16-byte) input word string are m_1, m_2, ...m_n then the output $Y2$ would be

$Y2 = 1$;

For i=1 to n

If ($m_i \geq maxrange$) then

$$Y2 = (k \times Y2 + (p\text{-}1)) \bmod p$$

$$Y2 = (k \times Y2 + (m_i - (2^{64} - p))) \bmod p$$

else

$$Y2 = (k \times Y2 + m_i) \bmod p$$

Return $Y2$

The operations in Layer 2 are somewhat complex. The main operations that have to be considered in hardware implementation are multiplication between 64-bit and 128-bit numbers in modulo ($2^{64} - 59$), multiplication of two 128-bit numbers in modulo ($2^{128} - 159$), 64 and 128-bit addition, and logical AND operation. The details are discussed in section 6.2.

Layer 3: Inner Product Hashing

The 16-bit output of Layer 2 is hashed into a 4-byte result using a simple inner product hash with affine translation. The prime modulus of ($2^{36} - 5$) is used to improve

security [75]. Two keys, $K_{L3\text{-}1}$ and $K_{L3\text{-}2}$, of length 64 bytes and 4 bytes respectively, are used for the inner product. The 16-byte input and 64-byte key ($K_{L3\text{-}1}$) are broken into 8 chunks as follows. Here M_i and k_i will be 2 and 8 bytes respectively.

For $i = 1$ to 8

$$M_i = M\,[(i\text{-}1)\times 2 + 1, \ldots., i\times 2]$$

$$k_i = K_{L3\text{-}1}\,[(i\text{-}1) \times 8 + 1, \ldots., i\times 8]$$

The inner product hashing is defined as follows.

$$Y3 = \{[[(M_1 \times k_1 + \ldots + M_8 \times k_8) \bmod (2^{36} - 5)] \bmod 2^{32}]\} \text{ XOR } K_{L3\text{-}2}$$

Since layer 3 output is only 4 bytes long, multiple iterations of the three layers are used with different keys each time until the UMAC output length is obtained. For example, in 32-bit implementation UHASH has to be done twice to get 8-byte output. Figure 55 shows the Layer 3 input and output details.

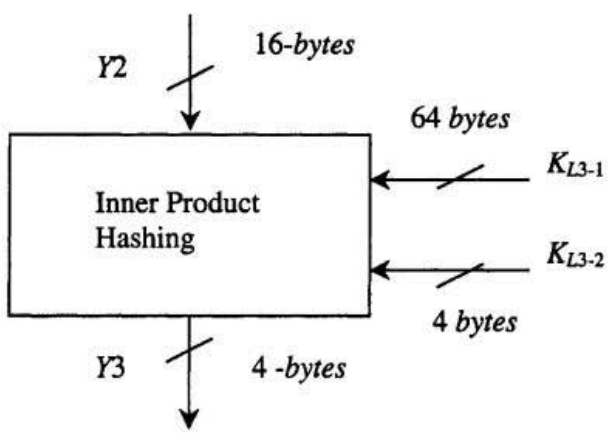


Figure 55. UHASH Layer 3

Layer 3 has multiplications between 16-bit and 64-bit numbers in modulo 2^{32} and modulus operation on prime number ($2^{36}-5$) as major operations. In addition to those the trivial operations such as mod 2^{32} and logical XOR operation are also involved.

6.2 Hardware complexity of UMAC

UMAC has primarily been targeted for software implementation. Most of the operations have been structured to suit some enhanced architectural features of modern processors. However, due to the attractive features such as the ability to trade off speed and security and the flexibility, it is worthwhile studying the suitability of UMAC for hardware implementation. The operations of each layer are summarized in Table 8.

The block length in bytes gives the minimum storage required during run time environment for the UMAC internal keys. UMAC has been designed to allow implementations which accommodate "on-line" authentication where the pieces of the message may be presented to UMAC at different times (but in correct order) and an on-line implementation will be able to process the message correctly without the need to buffer more than a few dozen bytes of the message.

Operations involved in Layer 1 are straightforward in hardware except the 32-bit multiplication. The Layer 1 has to be repeated until the full length of the message is hashed. The throughput of this layer can be easily improved by pipelining.

Table 8. Main operations in the three layers of UHASH.

	Layer 1	Layer 2	Layer 3
32 – bit addition (mod 2^{32})	✓		
64-bit addition	✓	✓	
32-bit multiplication (mod 2^{64})	✓		
128-bit × 64-bit multiplication		✓	
128-bit × 128-bit multiplication (for long messages)		✓	
Modulo (2^{64}-59)		✓	
Modulo (2^{128}-159) (for long messages)		✓	
Logical AND or XOR		✓	✓
128-bit addition			✓
Modulo (2^{36}-5)			✓
16-bit × 64-bit multiplication			✓

In Layer 2 there is 128-bit multiplication, which could be expensive in terms of speed. In Layer 3 too there are 16-bit and 36-bit multiplications. The other operations are not critical in terms of the speed. The modulo 2^{32} or 2^{64} can be easily implemented by merely ignoring any bits beyond the least significant 32 or 64, respectively. The modulus operations under a prime number as p such as (2^{64}-59) can be implemented without

division in following manner. Consider X as a number larger than 64 bits, which can be represented as

$$X = a2^{64} + b$$

In the field of the prime number p (which is $(2^{64}-59)$),

$$2^{64} - 59 = 0, \text{ hence}$$

$$X \bmod p = b + 59a$$

Hence, it is possible to carry out these modular operations without any division.

With respect to the area utilization of the UMAC, it is useful to consider the possible area consumption of the components that would be instantiated by the implementation. When a high end FPGA device such as Virtex V1000FG680 is targeted, the area utilizations (without optimization) of some major components are as follows. Here the percentage of slice utilization for each component is given.

Component	% Slice Utilization
32 – bit addition (mod 2^{32})	0.13 %
64-bit addition (mod 2^{64})	0.26%
32-bit multiplication	4.4 %
128-bit × 64-bit multiplier	8 %
128-bit × 128-bit multiplier	34.20 %
Modulo (2^{64}-59)	1.20 %
Modulo (2^{128}-159)	2.97 %
Modulo (2^{36}-5)	0.46 %

16-bit × 64-bit multiplier	4.03 %
128-bit addition	0.52 %
Logical AND	<< 1 %
Logical XOR (16 bit)	<< 1 %

The most expensive operation of the algorithm is the 128-bit multiplication which is carried out for messages of length > 2^{17}. Xilinx core generator library modules were parameterized to create 32-bit and 64-bit addition and 32-bit multiplication. The other operations were coded in VHDL to be implemented using the device resources. The throughput of the algorithm can be improved by introducing both parallel operations and pipelining. Since the operations of different message blocks are independent, it is possible to perform them in parallel by avoiding resource sharing. Layer 1 operations can be easily pipelined to improve the throughput. In this case some of the device resources have to be allocated for pipelining. This implies that the device will be fully utilized for UMAC hash operation. The key generation employs Rijndael, which has an area utilization of the 43 % of slices [51]. This has to be implemented separately. The evaluation of the hardware speed requires a full implementation of the design, which is beyond the scope of this thesis.

6.3 Conclusion

The UMAC is a complicated algorithm for hardware implementation. An efficient (fast) implementation in hardware could be very expensive in terms of complexity and

thus the FPGA device required. Although the algorithm can be pipelined and the operations are parallelizable, those advantages cannot be easily achieved with the constraints of the device resources in many of the today's popular FPGA devices. Although it might be implemented on a large FPGA device or using ASIC technology, the most critical operations have to be fully optimized to improve the throughput. This again could increase the resource utilization. However, because of the novel features and the efficiency in software, it is worthwhile to study the ways of adopting UMAC for hardware implementation.

Chapter 7

Conclusions

In many commercial applications protecting the integrity of information is even more important than its secrecy. With the advent of public key cryptography, digital signature schemes and Internet security schemes, cryptographic hash functions have gained much more prominence. In many applications the performance of cryptographic operations is a crucial factor, as it often becomes a bottleneck. As well, many security services handle a large number of security associations. Therefore key agility and algorithm agility are important issues in this context. Hence, in high-speed applications that employ the above-mentioned schemes, hardware encryption and authentication have become essential to meet these performance requirements. When these factors are considered, FPGA devices are a promising alternative for implementing cryptographic algorithms. In this study, two of the most common cryptographic hash algorithms and a hashed message authentication algorithm were implemented using FPGAs. Two hardware architectures, iterative and full loop unrolling, were investigated and efforts were taken to optimize the performance results of the designs. Some of the new techniques of message authentication were also investigated. In this chapter, the summary of this thesis, the conclusion and suggested future work are discussed.

7.1 Summary and Conclusions of the Study

The background of this research, which motivated towards this study, was first discussed. After a brief overview of IPSEC, which is the main area of application of MD5, SHA-1 and HMAC, different constructions of hash functions and message authentication codes were investigated. Some of the constructions, which are more suitable for hardware implementation, were also discussed. Followed by the descriptions of the constructions of MD5, SHA-1 and HMAC, the main issues related to hardware implementation were explored. In particular the overview of FPGAs and their applications were discussed. As the FPGA technology is a growing area that has a potential to provide performance benefits of ASICs and the flexibility of processors, it was selected as the target device for these implementations. FPGAs allow application specific hardware circuits to be created on demand to meet the requirements of a design. Moreover these hardware circuits can be dynamically modified partially or completely in time and space.

For both MD5 and SHA-1 implementations, two architectures were used: the iterative and full loop unrolled. As expected the iterative design provided the most area-optimized solution whereas the full loop unrolled design offered the most speed optimized solution. MD5 iterative design with double buffering offered about 185 Mbps throughput. High efforts in synthesizing and implementation were used along with "period" timing constraint for optimum speed results. As the area utilization was significantly low, several designs could be implemented on the same device and process multiple messages in parallel with proper management of I/O ports. The full loop

unrolled design of MD5, which was also implemented with double buffering and the same optimizing parameters, gave a throughput of 486 Mbps. According to the area utilization, at least two design modules could be accommodated in the device and process two messages in parallel.

SHA-1 full loop unrolled design with double buffering gave a throughput of 565 Mbps. As the area utilization was fairly high only one design can be fitted into a single device. The iterative design of SHA-1 showed a throughput of 121 Mbps. In this design the usage of memory was considerably high and the design is more complex than MD5.

The HMAC was implemented using SHA-1 full loop unrolled design as the base hash function. The maximum throughput obtained was 485 Mbps. As a consequence of these implementations, it was evident that the FPGAs were suitable to implement hash algorithms and hash based message authentication codes. The performance results meet some of the currently available IP bandwidths. Hence, these FPGA implementations can be used as components in cryptographic accelerators for use in IPSEC and other applications.

The size of the message has a considerable impact on the performance of cryptographic hash algorithms as padding has to be carried out all the time even if the message has a length of multiples of 512 bits. This becomes a large overhead for small messages especially in case of HMAC algorithm. As the Internet is one of the main areas of application of cryptographic hash functions, the understanding of Internet traffic is useful to study the performance of authentication algorithms. This was investigated using four traffic models. According to these models the average times of MAC calculations of

the FPGA implementations of HMAC and CBC-MAC-AES did not differ significantly. Particularly, the average times were almost the same as the size of the messages became larger. The software results showed that the timing performances of HMAC-SHA-1 and CBC-MAC-AES did not have a significant difference. As observed in FPGA implementations HMAC offers better timing performance as the message becomes larger.

As most of today's cryptographic algorithms follow a sequential structure, pipelining cannot be adopted to optimize the throughput. Hence, many of them cannot meet today's high-speed requirements. A recently reported approach that addresses this issue was then investigated. The universal message authentication algorithm that has been proposed by researchers was analyzed for hardware performance. It was shown that the complexity of the algorithm causes high resource utilization, and it would not be feasible for hardware implementation using existing FPGA devices.

7.2 Suggestions for Future Work

According to the area utilization of MD5, several design modules could be accommodated in the same device and process multiple messages in parallel. This would be an interesting effort for further investigation. In these implementations only the "period" timing constraint was used. This constraint covers only timing paths that start and end at a flip-flop, latch or synchronous RAM, which is clocked by a referenced net. It does not cover paths to output pads. By using other constraints such as "offset" it might be possible to improve the performance further. Therefore, implementation trials with other timing constraints are recommended. The use of a more recently introduced FPGA

device would also enhance the timing performance. However, as the structure of MD5 and SHA-1 does not allow parallelization, it is hard to achieve high throughput results. Two possible alternatives will be either to modify the general structure of cryptographic hash algorithms or to go for parallelizable hash algorithms with operations that are suitable for hardware implementation.

For the former alternative, several methods have been suggested by other researchers. One such methods is to use interleaved block chained digest structure. This replaces the original linear block chain with a finite number of block chains. A pre-determined finite number of chains are processed from independent initial values, such that the i^{th} block is part of the "i mod k"th chain. The resulting sequence of k digests forms another message, which can be hashed as a single block. There are several other proposals, which support parallelism in the general structure of the cryptographic hash algorithms. However, these proposals need further analysis for their security properties.

For the latter option, some of the members of the universal hash function family that have suitable properties for hardware implementation could be utilized. These have to be further investigated for the optimum hardware implementations as well as the optimum cryptographic properties. Though the proposed UMAC would be too large for the existing FPGA devices, it may be suitable for ASIC implementation. The complete implementation of UMAC on hardware is recommended so as to investigate its speed performance.

References

[1]. G. J. Simmons, *Contemporary Cryptology: The Science of Information Integrity,* Piscataway, NJ, IEEE Press, 1992

[2]. Organization for Economic Co-Operation and Development (OECD) Guidelines, "*Guidelines on the Protection and Transborder Flows of Personal Data*", available at http://www1.oecd.org/dsti/sti/it/secur/prod/PRIV-EN.HTM#2.

[3]. M. Bellare, R. Canetti and H. Krawczyk, "Keying Hash Functions for Message Authentication," in *proceedings of Advances in Cryptology- CRYPTO'96*, Lecture Notes in Computer Science Vol. 1109, Springer-Verlag, pp. 1-15, 1996.

[4]. R. Rivest, "*The MD5 Message-Digest Algorithm*," IETF Network Working Group, RFC 1321, April 1992.RFC 1321.

[5]. E. Hong, J.H. Chung and C.H.Lim, "Hardware Design and Performance Estimation of the 128-bit Block Cipher CRYPTON," in *Proceedings of first International Workshop, CHESS'99*, Lecture Notes in Computer Science1717, Springer-Verlag, pp. 49-61, 1999

[6]. S. Kent and R. Atkinson, "*Security Architecture for the Internet Protocol*," IETF Network Working Group, RFC 2401, November 1998.

[7]. Cylan IP Security white paper, 1997, available at http://www.cylan.com/files/whpaper.htm.

[8]. FIPS PUB 180-1, "*Secure Hash Standard*," Federal Information Processing Standard (FIPS), Publication 180-1, National Institute of Standards and Technology, US Department of Commerce, Washington D.C., April 1995.

[9]. S. Kent and R. Atkinson "*The Use of HMAC-MD5-96 Within ESP and AH*," IETF Network Working Group, RFC 2401, November 1998.

[10]. C. Madson and R. Glenn, "*The Use of HMAC-SHA-1-96 Within ESP and AH*," IETF Network Working Group, RFC 2404.

[11]. D. Whiting and Schneier, "*Improved Twofish Implementations*" Twofish Technical Report #3, available at http://www.counterpane.com.

[12]. Cisco Systems Inc, IPSEC white paper, available at http://www.pipelinks.com/ warp/ public/cc/techno/prtocol/ipsecure/ipsec/tech/ipsec_wp.htm.

[13]. S. Kent and R. Atkinson, " *IP Authentication Header*" IETF Network Working Group, RFC 2402, November 1998.

[14]. S. Kent and R. Atkinson, " *Internet Protocol- DARPA Internet Protocol Specification*" IETF Network Working Group, RFC 791, September 1981.

[15]. S. Deering and R. Hinden, "*Internet Protocol, Version 6 (IPV6) Specification*," IETF Network Working Group, RFC 2460, December 1998.

[16]. W. Stallings, *Cryptography and Network Security*, Second edition, Upper Saddle River, NJ: Prentice Hall, 1997.

[17]. S. Kent and R. Atkinson, "IP Encapsulating Security Payload (ESP)," IETF Network Working Group, RFC 2406, November 1998.

[18]. ANS Glossary 2000, available at http://www.its.bldrdoc.gov/projects/t1glossary 2000/_hash_function.html.

[19]. D. Stinson, *Cryptography: Theory and Practice*, Boca Raton, FL, CRC Press, 1995.

[20]. FIPS 186-2, "*Digital Signature Standard (DSS)*" Federal Information Processing Standard (FIPS), Publication 186-2, National Institute of Standards and Technology, US Department of Commerce, Washington D.C., January 2000.

[21]. FIPS 186, "*Digital Signature Standard (DSS)*" Federal Information Processing Standard (FIPS), Publication 186, National Institute of Standards and Technology, US Department of Commerce, Washington D.C., May1994.

[22]. B. Preneel, R. Govaerts and J. Vandewalle, "Hash Functions Based on Block Ciphers: A Synthetic Approach," in *proceedings of Advances in Cryptology-CRYPTO'93*, Lecture Notes in Computer Science Vol. 773, Springer-Verlag, pp. 368-378, 1996.

[23]. S. Bakhtiari , R. Safavi-Naini and J. Pieprzyk, *Cryptographic Hash Functions: A Survey*, Technical Report 95-09, Department of Computer Science, University of Wollongong, July 1995.

[24]. R. L. Rivest, A Shamir and L. Adleman, "A Method for Obtaining Digital Signatures and Public Key Cryptosystems" Communications of ACM, vol. 21, pp. 120-126, 1978.

[25]. B. Preneel, "The State of Cryptographic Hash Functions," in proceeding of *Lectures of Data Security*, Lecture Notes in Computer Science vol. 1561, Springer-Verlag, pp. 158-182, 1999.

[26]. R. Rivest, "The MD4 Message Digest Algorithm," IETF Network Working Group, RFC 1320, April 1992.

[27]. I. B. Damgard, "A Design Principle for Hash Functions," in *proceedings of Advances in Cryptology-CRYPTO'89*, Lecture Notes in Computer Science vol. 435, Springer-Verlag, pp. 416-427, 1989.

[28]. P. Camion and J. Patavin, "The Knapsack Hash Function Proposed at CRYPTO'89 Can be Broken" in *proceedings of Advances in Cryptology-EUROCRYPT'91*, Lecture Notes in Computer Science vol. 576, Springer-Verlag, pp. 39-53, 1991.

[29]. S. Wolfram, "Random Sequence Generation by Cellular Automata" in *proceedings of Advances in Applied Mathematics*, vol 7, pp. 123-169, 1986.

[30]. J. Daeman, R. Govaerts and J Venewalle, "A Hardware Design Model for Cryptographic Algorithms," in proceedings of Computer Security-ESORICS'92, Lecture Notes in Computer Science vol. 648, Springer-Verlag, pp. 419-434, 1992.

[31]. X. Lai, R.A. Rueppel, and J. Woollven, "A fast cryptographic checksum algorithm based on stream ciphers," in *proceedings of Advances in Cryptology - AUSCRIPT '92*, Lecture Notes in Computer Science, Springer-Verlag, pp.339-348, 1992.

[32]. M.Bellare, J. Kilian and P. Rogaway, "The Security of the Cipher Block Chaining Message Authentication Code," in *journal of computer and System Sciences*, Vol. 61, No. 3, Academic Press, pp. 362-399, 2000.

[33]. ANSI X9.9, American National Standard for Financial Institution Message Authentication (wholesale), American Bankers Association, 1981-Revised 1986.

[34]. FIPS 113, " *Computer Data Authentication*," Federal Information Processing Standard (FIPS), Publication 113, National Institute of Standards and Technology, US Department of Commerce, Washington D.C., 1985.

[35]. M. Bellare, J. Kilian and P. Rogaway, " The Security of Cipher Block Chaining," in *proceedings of Advances in Cryptology- CRYPTO'94*, Lecture Notes in Computer Science vol. 839, Springer-Verlag, pp. 340-358, 1994.

[36]. M. Bellare, R. Guvaerin and P. Rogaway, "XOR MACs: New Method for Message Authentication Using Finite Pseudorandom Functions," in *proceedings of Advances in Cryptology- CRYPTO'95*, Lecture Notes in Computer Science vol. 963, Springer-Verlag, pp. 15-28, 1995.

[37]. G. Tsudik, "Message Authentication with One-Way Hash Functions" *ACM Computer Communications Review*, vol. 22, No. 5, pp. 29-38, 1992.

[38]. J. Givin, K. McCiognrie and J. Davin, "Secure Management of SNMP Networks," IETF Network Working Group, RFC1157 1990.

[39]. B. Preneel and P.C. VanOorschot, "MD_x-MAC and Building Fast MACs from Hash Functions," in *proceedings of Advances in Cryptology- CRYPTO'95*, Lecture Notes in Computer Science vol. 963, Springer-Verlag, pp. 1-14, 1995.

[40]. S. Bakhtiari, S. Safavi-Nani and J. Pieprzyk, "Practical and Secure Message Authentication," Series of Annual Workshop on Selected Areas in Cryptography (SAC'95), pp. 55-68, May 1995.

[41]. J. L. Carter and M. N. Wegman, "Universal Classes of Hash Functions," *Journal of Computer and System Services*, vol. 18, pp. 143-154, 1979.

[42]. J. L. Carter and M. N. Wegman, "New Hash Functions and Their Use in Authentication and Set Equality," *Journal of Computer and System Services*, vol. 22, pp. 265-279, 1981.

[43]. J. Black Jr., "Message Authentication Codes," PhD Thesis, University of California Davis, California, USA, 2000. PhD Thesis 2000.

[44]. P. Rogaway, "Bucket Hashing and Its Application to Fast Message Authentication," in *proceedings of Advances in Cryptology-CRYPTO'95*, Lecture Notes in Computer Science vol. 963, Springer-Verlag, pp. 29-42, 1995.

[45]. S. Helvi and H. Krawczyk, "MMH: Software Message Authentication in the Gbit/second Rates," in *proceedings of 4th workshop on Fast Software Encryption*, Lecture Notes in Computer Science vol. 1267, Springer-Verlag, pp. 172-189, 1997.

[46]. J. Black, S. Halevi, A. Hevia, H. Krawczyk, T. Krovetz and P. Rogaway, "UMAC-Fast and Secure Message Authentication," in *proceedings of Advances in Cryptology-CRYPTO'99*, Lecture Notes in Computer Science vol. 1666, Springer-Verlag, pp. 216-233, 1999.

[47]. K. Ohta and K. Koyama, "Meet-in-the-Middle Attack on Digital Signature Schemes," in *proceedings of Advances in Cryptology- AUSCRYPT'90*, Lecture Notes in Computer Science vol. 453, Springer-Verlag, pp. 140-154, 1990.

[48]. J. Pieprzyk and B. Sadeghiyan, "Design of Hash Algorithms," Lecture Notes in Computer Science vol. 756, Springer-Verlag, 1993.

[49]. FIPS PUB ZZZ, "Advanced Encryption Standards (AES)," Federal Information Processing Standard (FIPS), Publication AES Draft, National Institute of Standards and Technology, US Department of Commerce, Washington D.C., 2001.

[50]. M. Riaz and H.M. Heys, "The FPGA Implementation of the RC6 and CAST-256 Encryption Algorithms", in *Proceedings of IEEE Canadian Conference on Electrical and Computer Engineering CCECE '99*, Edmonton, Alberta, May 1999.

[51]. A. Elbirt, W. Yip, B. Chetwynd and C. Paar, "An FPGA Implementation and Performance Evaluation of the AES Block Cipher Candidate Algorithm Finalists," in *proceedings of 3rd AES Candidate Conference*, available at http://www.nist.gov/aes.

[52]. A. Dantalis, V. K. Prasanna and J. D. P. Rolim, "A Comparative Study of Performance of AES Final Candidates Using FPGAs," in *proceedings of 3rd AES Candidate Conference*, available at http://www.nist.gov/aes.

[53]. K. Gaj and P. Chodoxiec, "Comparison of the Hardware Performance of the AES Candidates Using Reconfigurable Hardware," in *proceedings of 3rd AES Candidate Conference*, available at http://www.nist.gov/aes.

[54]. R. R. Taylor and S. C. Goldstein, "A High-Performance Flexible Architecture for Cryptography," in *proceedings of Cryptographic Hardware and Embedded Systems,* Lecture Notes in Computer Science vol. 1717, Springer-Verlag, pp. 231-245, 1999.

[55]. A. Dewey, *Analysis and Design of Digital Systems with VHDL*, PWS Publishing Company, 1997.

[56]. M. Smith, *Portions from Application-Specific Integrated Circuits*, available at http://www-ee.eng.hawaii.edu/~msmith/ASICs/HTML/Book2/.

[57]. S. Brown and J. Rose, "FPGA and CPLD Architectures: A Tutorial," in *proceedings of IEEE Design and Test of Computers*, vol. 12, No. 2, pp. 42-57, 1996.

[58]. FIPS PUB # HMAC, " *The Keyed Hash Message Authentication Code (HMAC)*," Federal Information Processing Standard (FIPS) Publication # HMAC, National Institute of Standards and Technology, US Department of Commerce, Washington D.C., 2001.

[59]. Xilinx Virtex 2.5 V programmable date arrays product specification, available at http://www.xilinx.com/products/virtex.

[60]. Xilinx home page: http://www.xilinx.com.

[61]. J Case, N. Gupta, J. Mittal and D. Ridgeway, "*Design Methodologies for Core-Based FPGA Designs*," Xilinx white paper available at http://www.xilinx.com/ products/logicore/core_papers.htm.

[62]. S. Yalamanchili, *Introductory VHDL: from Simulation to Synthesis*, Prentice Hall, Upper Saddle River, NJ, 2001.

[63]. Foundation Series 3.1i Quick Start Guide, available at http://toolbox.xilinx.com /docsan/3_1i/data/fndtn/fqs/chap02/fqs02000.htm.

[64]. Xilinx records #2703, available at http://support.xilinx.com/xlnx/xil_ans_display.jsp?iLanguageID=1&iCountryID=1&getPagePath=2703 1999.

[65]. R. C. Merkle, "One Way Hash Functions and DES" in *proceedings of Advances in Cryptology-CRYPTO'89*, Lecture Notes in Computer Science vol. 435, Springer-Verlag, pp. 428-446, 1989.

[66]. J. Touch, "Performance Analysis of MD5, in *proceedings of SIGCOMM'95*, Boston, pp 77-86, 1995.

[67]. H. Krawczyk, M. Bellare and R. Canetti, "*HMAC: Keyed Hashing Message Authentication*," IETF Network Working Group, RFC-2104, February 1997.

[68]. A. Feldmann, J. Rexford and R. Caceres, "Efficient Policies for Carrying Web Traffic Over Flow-Switched Networks," *IEEE/ACM transactions on Networking*, pp. 673-685, December 1998.

[69]. S. McCreary and K. Claffy, "Treds in Wide Area IP Traffic Patterns," A view from Ames International Exchange, Co-operative Association for International Data Analysis (CAIDA) Report, 2000, available at http://www.caida.org/outreach/ papers/AIX0005.

[70]. J. Black, S. Halevi, H. Krawczyk, T. Kovetz and P. Rogaway, "Update on UMAC Fast Message Authentication" available at http://www.cs.ucdavis.edu/~rogaway/umac.

[71]. J. Black and P. Rogaway, " CBC MACs for Arbitrary-Length Messages: The Three-Key Constructions," in proceedings of advances in Cryptology-

CRYPTO'00, Lecture Notes in Computer Science, Springer-Verlag vol. 1880, pp. 197-215, 2000

[72]. NIST home page, "AES Algorithm (Rijndael) Information", available at http://csrc.nist.gov/encryption /aes /rijndael/.

[73]. "Sha1.c: Implementation of the Secure Hash Algorithm", November, 2000, available at http://www.di-mgt.com.au/src/sha1.c.txt.

[74]. V. Shoup " On fast and Provably Secure Message Authentication Based on Universal hashing," in *proceedings of advances in Cryptology-CRYPTO'96*, Lecture Notes in Computer Science, Springer-Verlag vol. 1109, pp. 313-328, 1996.

[75]. T. Krovetz, J. Black, S. Halevi, A. Hevia and H. Krawczyk, P. Rogaway, " *UMAC: Message Authentication Code Using Universal hashing*," IPSec Working Group, Internet-Draft, October 2000.

Appendix A

Results of Timing Simulation with Back Annotation.

This appendix shows the timing simulation results of MD5, SHA-1 and HMAC-sha-1. For this the Standard Delay Format (SDF) file which is created during the implementation is used. The SDF file contains the timing details of the design that have to be used during the back annotation. This has to be invoked with the timing simulation model. The timing simulation model (VHDL file) is created from the Native Generic Database (NGD) file by running NGD2VHDL. The same test benches utilized for behavioral and functional simulation were used.

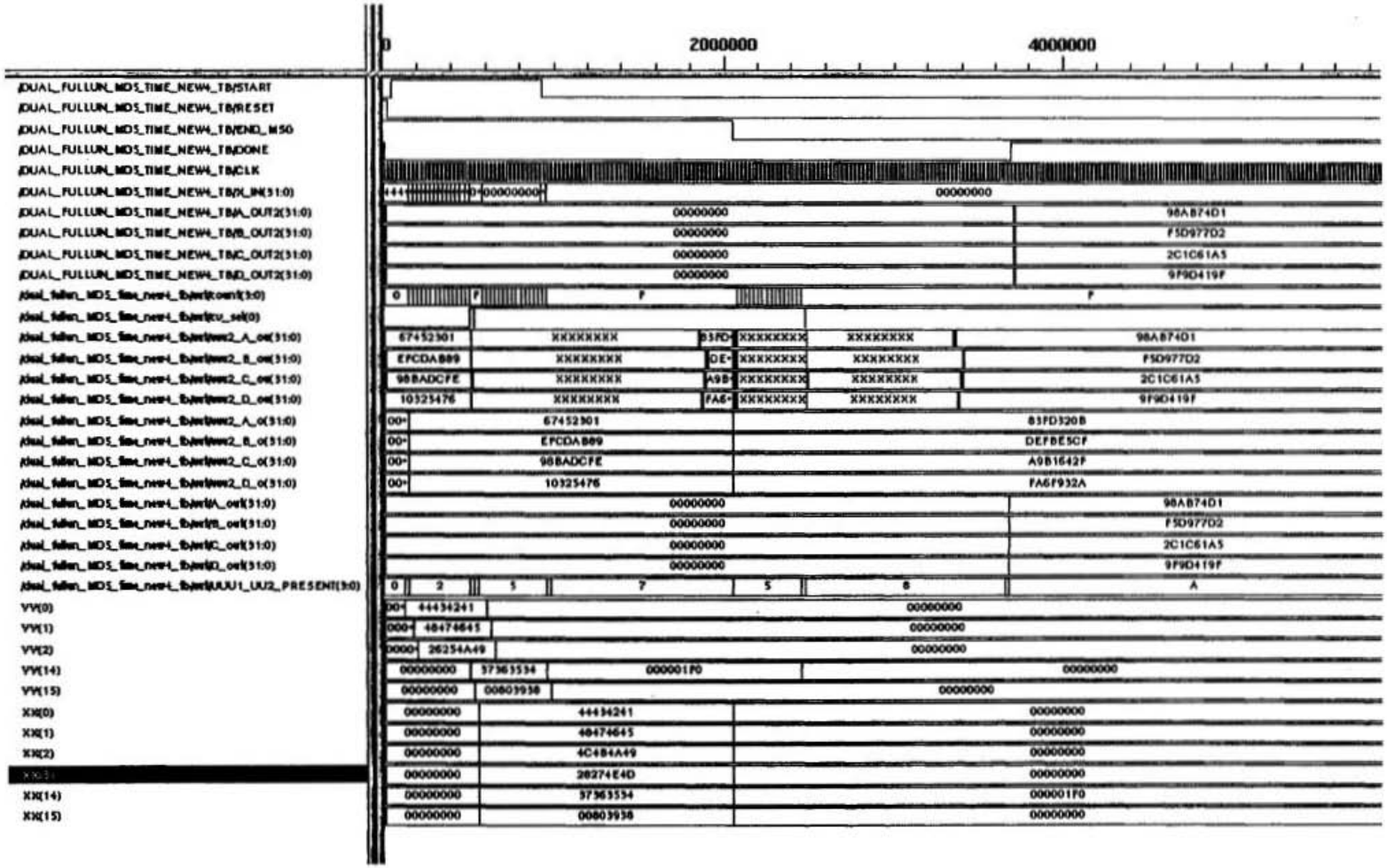


Figure A1. Timing simulation MD5_full loop unrolled design

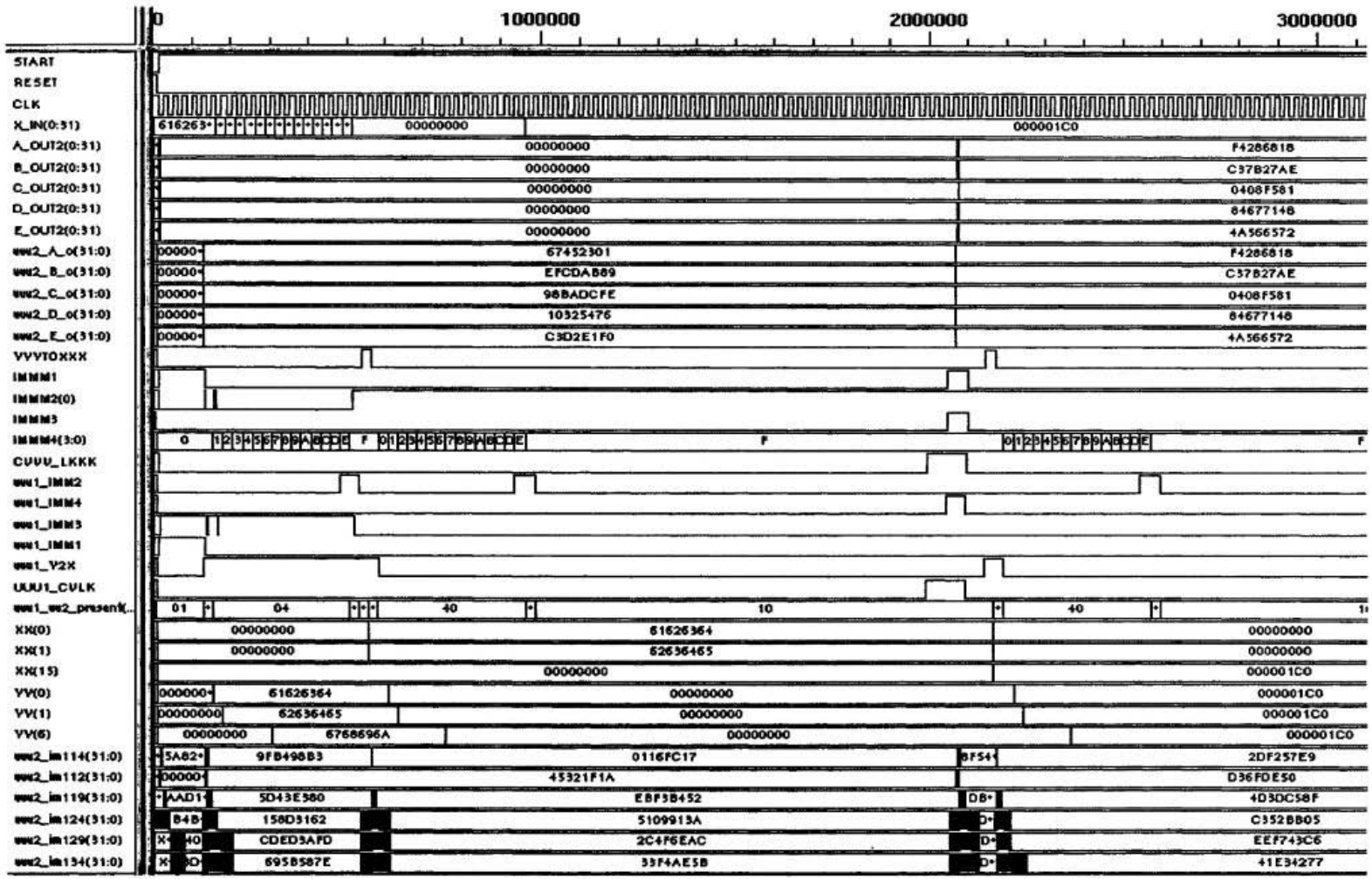


Figure A2. Timing simulation of SHA-1 full loop unrolled design

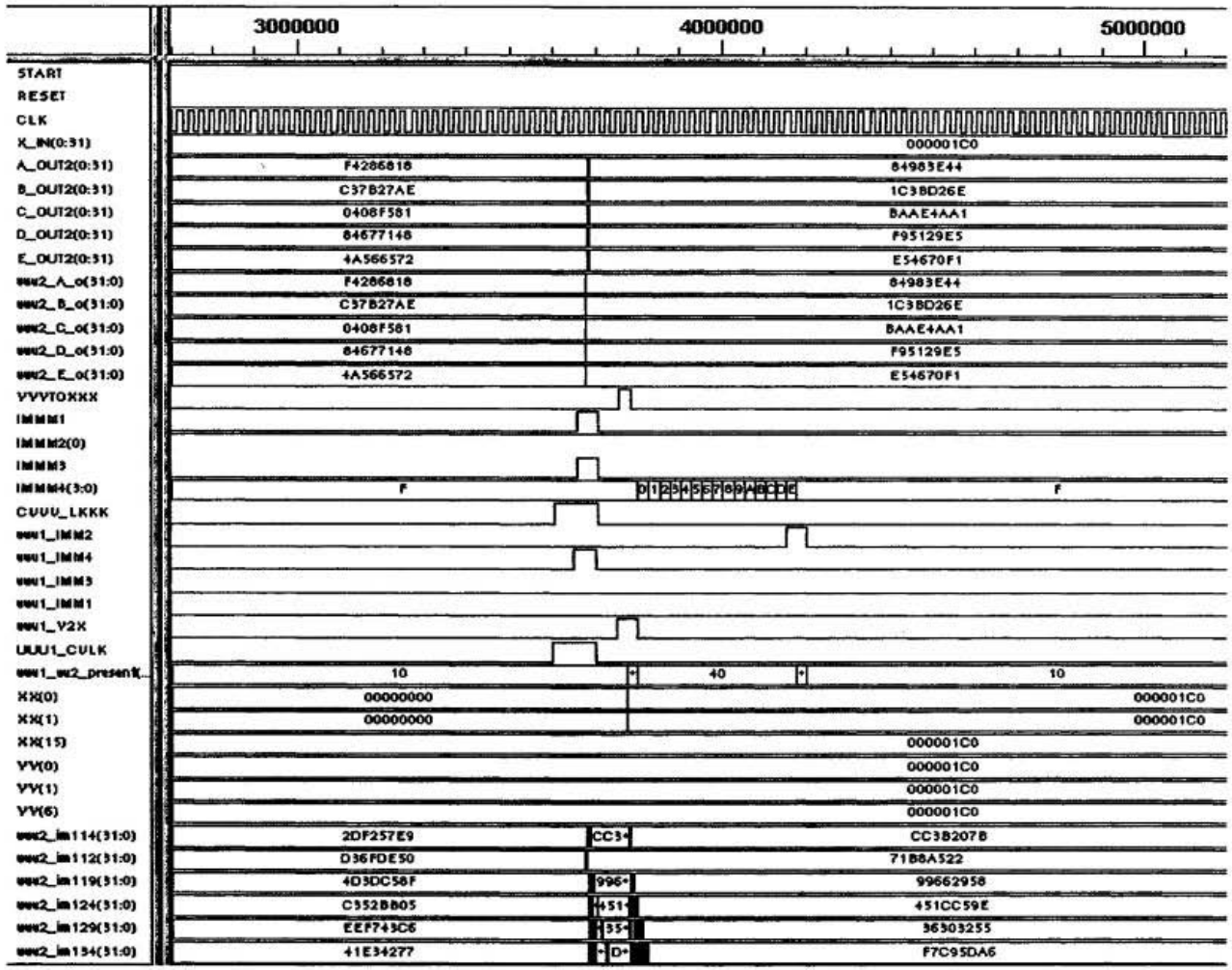


Figure A2. Timing simulation of SHA-1 full loop unrolled design (Contd)

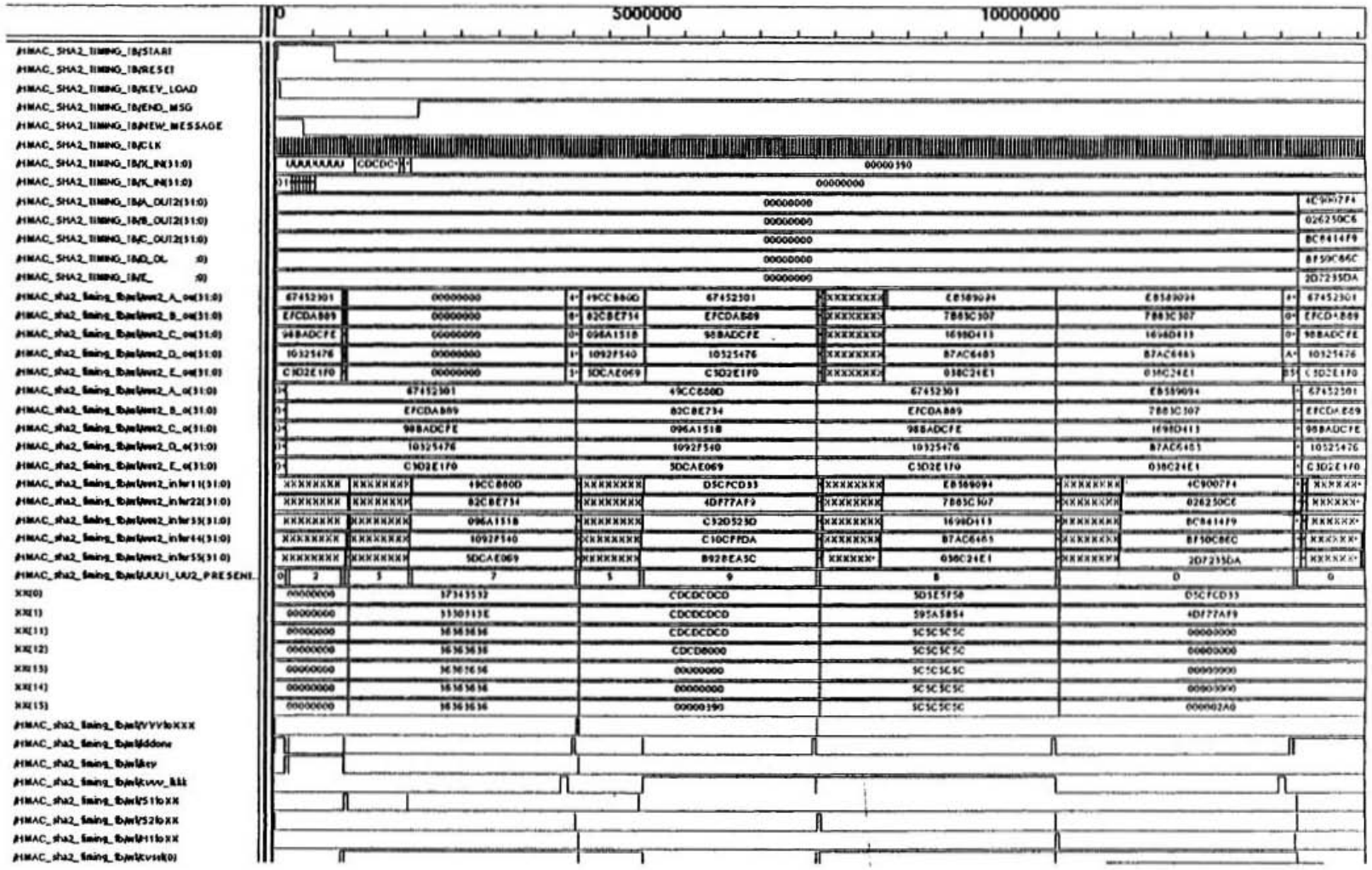


Figure A3. HMAC-sha-1 timing simulation

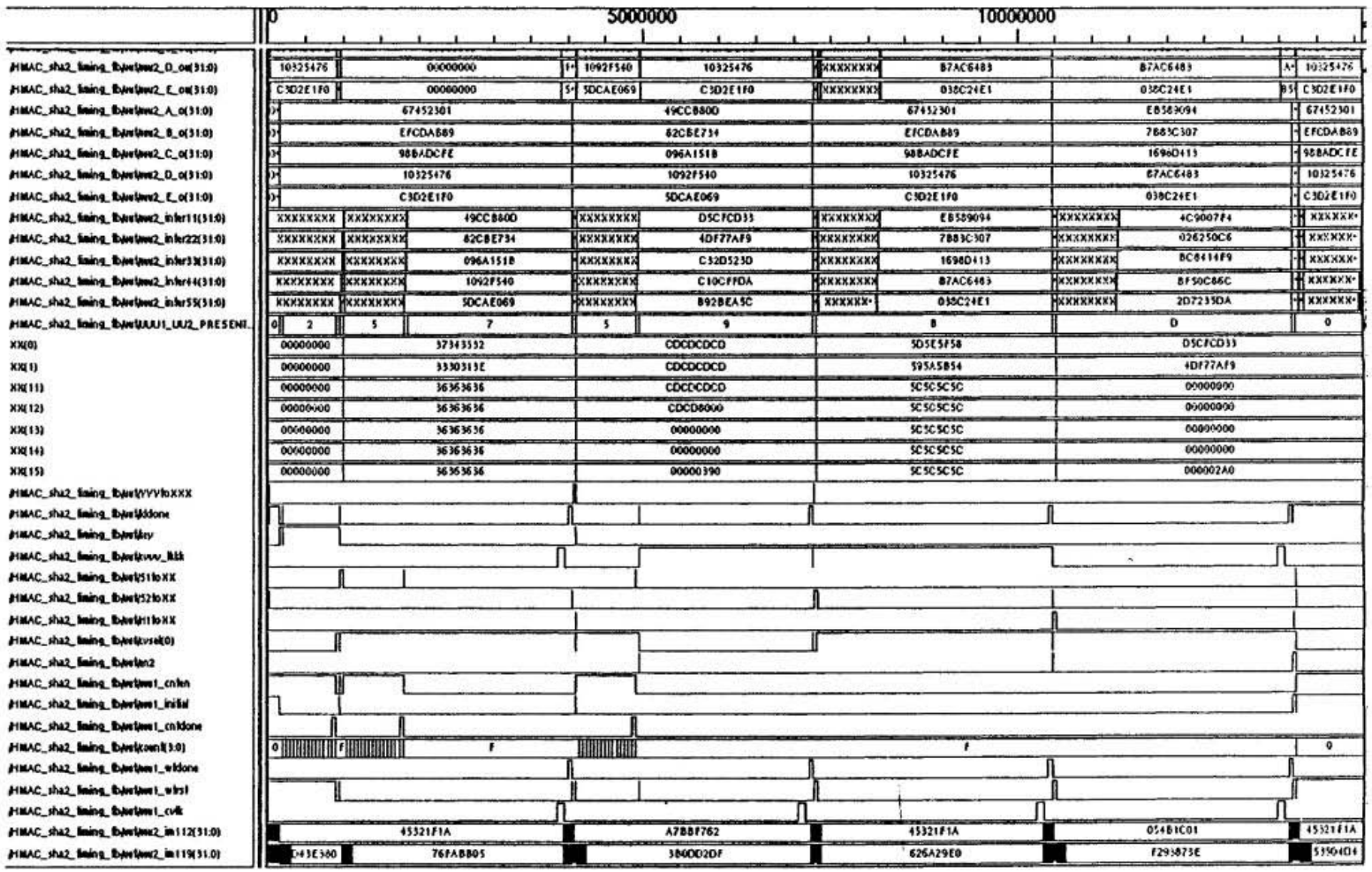


Figure A3. HMAC-sha-1 timing simulation (Contd)

Appendix B

This appendix gives the read and write operations of the RAM set up during the 80 steps of SHA-1 iterative design. During the loading time, the message block is written in all the 8 RAM modules. Then the read and write operations of each RAM module vary according to Table A1. The RAM set up for this is given in Figure 25. While reading the initial 16 words, the next words (17^{th} word and onwards) can be calculated by XORing four words from the previous 16 words together. In this case four words have to be read at a time and the calculated word has to be stored in a suitable RAM to prevent any clash between read and write operations among the modules in future steps. The notations used in the table are as follows.

ReadX Reading the word for step X.

R# Reading memory address # of a RAM to calculate a future word.

W# Writing of a calculated word at memory location # of a selected RAM

Step	RAM_ A	RAM_B	RAM_A1	RAM_B1	RAM_A2	RAM_B2	RAM_A3	RAM_B3
0	read0	W16		R2		R13	W16	R8
1	read1	R3	W17		W17		R14	R9
2	read2	R4	R10	W18		W18		R15
3	read3	R16	R5	R11	W19		W19	
4	read4		R17	R12	R6	W20		W20
5	read5	W21		R18	R7	R13	W21	
6		read6	W22		R19	R8	R14	W22
7	W23	read7		W23		R20	R9	R15
8	read8	R16	W24		W24		R21	R10
9	read9	R11	R17	W25		W25		R22
10	R23	read10	R12	R18	W26		W26	
11		R13	R24	read11	R19	W27		W27
12	W28		read12	R25	R14	R20	W28	
13		W29		Read13	R26	R15	R21	W29
14	W30		W30	Read14		R27	R16	R22
15	R23	W31	R17		W31		R28	
16		read16	R24	W32		R18		R29
17	R30		read17	R25	W33		R19	
18		R31		Read18	R26	W34		R20
19		R21	W35	R32	read19	R27	W35	
20			R22		R33	read20	R28	W36
21	W37			R23		R34	read21	R29
22	R30	W38			R24		R35	read22
23	read23	R31	W39	R25		W39		R36
24	R37		read24	R32	W40		R26	
25		R38		read25	R33	W41		R27
26	R28		R39		read26	R34	W42	
27		R29			R40	read27	R35	W43
28	W44		R30	W44		R41	read28	R36
29	R37	W45			R31		R42	read29
30	read30	R38	W46	R32				R43
31		read31	R39	R44	R33			
	W47							
32		R45		Read32	R40	R34		
33	R35		W49	R41	read33	R46		W48
34		R36	R42		read34	R47		W50
35	W51		R43		W50	R37		R48
36		read36		R44	W51		R38	R49
37		R45	W52		R50	read37		R39
38	W53	R45		R40	R51		read38	
39	W54		R52	R41		R47		read39
40	R53	W55	R42	read40		R47		R48

Step	RAM_A	RAM_B	RAM_A1	RAM_B1	RAM_A2	RAM_B2	RAM_A3	RAM_B3
41	R54		R43	read41		W56		R49
42		R55	read42	R44	R50		W57	
43		R45	read43		R51	R56	W58	
44		W59	R52	read44		R46	R57	
45	R53	read45			W60	R47	R58	
46	R54	R59	W61			read46		R48
47		R55		W62	R60	read47		R49
48	W63		R61		R50	R56		read48
49			W64	R62	R51		R57	read49
50	R63		R52		read50	W65	R58	
51	R53	R59	R64	W66	read51			
52	R54		read52		R60	R65		W67
53	read53	R55	R61	R66				W68
54	read54			R62	W69	R56		R67
55	R63	read55			W70		R57	R68
56		W71	R64		R69	read56	R58	
57	W72	R59			R70	R65	read57	
58		R71	W73	R66	R60		read58	
59	R72	read59	R61			W74		R67
60			R73	R62	read60		W75	R68
61	R63		read61		R69	R74	W76	
62		W77	R64	read62	R70		R75	
63	read63	R71				R65	R76	W78
64	R72	R77	read64	R66				W79
65						read65		
66				read66				
67								read67
68								read68
69					read69			
70					read70			
71		read71						
72	read72							
73			read73					
74						read74		
75							read75	
76							read76	
77		read77						
78								read78
79								read79

Table B1. Read and write operations of the RAM set up of SHA-1 iterative design