Design and Performance Analysis of a Reconfigurable, Unified HMAC-Hash Unit for IPSec Authentication

by

Esam Ali Hasan Khan
B.Sc., King Fahd University of Petroleum and Minerals, 1999
M.Sc., King Fahd University of Petroleum and Minerals, 2001

A Dissertation Submitted in Partial Fulfillment of the Requirements
for the Degree of

DOCTOR OF PHILOSOPHY

in the Department of Electrical and Computer Engineering

Design and Performance Analysis of a Reconfigurable, Unified HMAC-Hash Unit for IPSec Authentication

by

Esam Ali Hasan Khan
B.Sc., King Fahd University of Petroleum and Minerals, 1999
M.Sc., King Fahd University of Petroleum and Minerals, 2001

Supervisory Committee

Dr. Fayez Gebali, (Department of Electrical and Computer Engineering)

Supervisor

Dr. Mostafa Abd-El-Barr, (Department of Electrical and Computer Engineering)

Co-Supervisor

Dr. Issa Traoré, (Department of Electrical and Computer Engineering)

Departmental Member

Dr. Frank Ruskey, (Computer Science Department)

Outside Member

Dr. M. Watheq El-Kharashi, (Department of Electrical and Computer Engineering)

Additional Member

Supervisory Committee

Dr. Fayez Gebali
Supervisor

Dr. Mostafa Abd-El-Barr
Co-Supervisor

Dr. Issa Traoré
Departmental Member

Dr. Frank Ruskey
Outside Member

Dr. M. Watheq El-Kharashi
Additional Member

ABSTRACT

In this dissertation, we discuss the design of a reconfigurable, unified HMAC-hash unit for IPSec authentication. The proposed unit is reconfigurable at runtime to enable implementing any of six standard algorithms: MD5, SHA-1, RIPEMD-160, HMAC-MD5, HMAC-SHA-1, and HMAC-RIPEMD-160. The designed unit can be used for IPSec or any other security application that uses hash functions, such as digital signature. We applied speedup techniques, such as pipelining and parallelism, to enhance the design of the HMAC-hash unit. We also proposed a key reuse technique to improve the HMAC throughput. We used an emerging system design methodology in designing the HMAC-hash unit. This methodology uses a high level language, Handel-C, to implement the designed unit and directly map it to FPGA platforms. We used the available constructs of Handel-C to conduct a design space exploration of the HMAC-hash unit. The performance of the designed unit was analyzed and compared to performance reported in previous work. To our knowledge, this work is the first in the literature that integrates six standard hash algorithms in one unified, reconfigurable unit. It is also the first in the literature that implements HMAC-RIPEMD-160 on FPGA. The work reported in this dissertation is the first to integrate HMAC with three hash functions. The achieved throughput is 173.69 Mbps for MD5 and 139.38 Mbps for each of SHA-1 and RIPEMD-160. Compared to results reported in

previous work, our unit achieves better throughput than those integrating three or more hash functions and a comparable throughput to those integrating two hash functions. We achieved better maximum frequency, which is 44.1 MHz, than all other work. We achieved comparable results to those integrating HMAC with some hash functions. The area utilization of the designed unit is less than 33% of the available logic on the FPGA chip we used. Thus, the designed unit can fit on a single FPGA chip as an SoC.

Table of Contents

List of Tables

List of Figures

List of Abbreviations

AES	Advanced Encryption Standard
AH	Authentication Header
ALP	Adaptive Logic Processor
ALU	Arithmetic and Logic Unit
ASIC	Application Specific Integrated Circuit
ASIP	Application Specific Instruction set Processor
CFHF	Collision Free Hash Function
COBRA	Cryptographic (Optimized for Block Ciphers) Reconfigurable Architecture
CRHF	Collision Resistant Hash Function
DES	Data Encryption Standard
DK	Design Kit
DSA	Digital Signature Algorithm
EDA	Electronic Design Automation
EDIF	Electrical Design Interchange Format
ESP	Encapsulated Security Payload
FIP	Fixed Instruction set Processor
FPGA	Field Programmable Gate Array
FreeBSD	Free Berkeley Software Design
FreeS/WAN	Free Secure Wide Area Network
FSM	Finite State Machine
GPP	General Purpose Processor
HAS	Hash function Algorithm Standard
HDL	Hardware Description Language

HMAC	Keyed-Hash Message Authentication Code
ICG	Initial Condition Generator
IDE	Integrated Development Environment
IDEA	International Data Encryption Algorithm
IETF	Internet Engineering Task Force
IKE	Internet Key Exchange
IOB	Input/Output Block
IP	Internet Protocol
IPSec	IP Security
ISAKMP	Internet Security Association and Key Management Protocol
ISE	Integrated Software Environment
IV	Initial chaining Value
LUT	Look Up Table
MAC	Message Authentication Code
MD5	Message Digest algorithm (version 5)
MDC	Modification Detection Code
MPRAM	Multi-Ported RAM
NAPA	National Semiconductor's Adaptive Processing Architecture
NIST	National Institute of Standards & Technology
NRE	Non Recurring Engineering
OWHF	One Way Hash Function
PDK	Platform Developer's Kit
PIN	Personal Identification Number
RAM	Random Access Memory
RC5	Rivest Cipher (version 5)
RFC	Request For Comment
RIPEMD	RACE Integrity Primitive Evaluation Message Digest
RNG	Random Number Generator

ROM	Read-Only Memory
RSA	Rivest, Shamir, and Adleman
RTL	Register Transfer Level (or Language)
RTR	Run-Time Reconfigurability
SA	Security Association
SHA-1	Secure Hash Algorithm 1
SoC	System-on-a-Chip
TAM	Test Access Mechanism
TDES	Triple DES (also called 3DES)
TTM	Time To Market
VHDL	VHSIC (very high speed integrated circuits) HDL
VPN	Virtual Private Network
WAP	Wireless Application Protocol

List of Notations

§	Section
0b	A prefix used with binary values
0x	A prefix used with hexadecimal values
$+$	Mathematical addition operation
$\&$	The bitwise AND operation
$\|$	The bitwise OR operation
$\oplus$	The bitwise XOR operation
$\overline{V}$	The logical complement of variable V
&&	The logical AND operation (used with conditions)
@	Concatenation operation
V @ $(0x00)_n$	padding of n bytes of zeros to the right of variable V
$Trunc_n(V)$	Truncation of variable V to the n leftmost bits.
abs	The absolute value
$ROTL^n(V)$	Rotate-left operation on variable V by n bits
mod	remainder of a division
MUX(X, Y, SEL)	Select X if SEL is correct, and select Y otherwise
MUX3(X, Y, Z)	Select X if MD5, Y if SHA-1, and Z if RIPEMD-160
sin	The sine function

Acknowledgement

"This is by the Grace of my Lord - to test me whether I am grateful or ungrateful! And whoever is grateful, truly, his gratitude is for (the good of) his own self; and whoever is ungrateful, (he is ungrateful only for the loss of his own self). Certainly my Lord is Rich (Free of all needs), Bountiful."

(Solomon): Holy Quarn, 27:40.

"And say: My Lord! Increase me in knowledge."

Holy Quarn, 20:114.

In the Name of Allah, the Most Gracious, the Most Merciful. All thanks are for Allah, who made this work complete, and whose graces are endless. All prayers and peace be upon Mohammed, the prophet and messenger of Allah.

Our prophet Mohammed (PBUH) said: "whoever does not thank people, will not thank Allah".

I could not achieve the work done in this dissertation without having a happy home life. My great thanks to my mother, who was patient with me being away from her and encouraged me to continue this path. My great appreciations to my dear wife, who made my home a comfortable place to work and supported me in all stages and times.

I would like here to express my great thanks to my supervisors, Dr. Fayez Gebali and Dr. Mostafa Abd-El-Barr. This work would not be accomplished without their guidance and advise. Their efforts are highly appreciated. Special thanks to Dr. Mohammed Watheq El-Kharashi, who worked very close with me to accomplish this work successfully.

I also would like to thank the other members of my supervisory committee: Dr. Issa Traoré and Dr. Frank Ruskey. I also would like to thank Dr. Ahmed Sourour, who was a member of the committee, but could not continue for health reasons. My great thanks to Dr. Hussein Alnuweiri, who agreed to act as an external examiner. I appreciate their kind agreement to read my dissertation and spend the time and effort to examine me.

My course of study was sponsored by the Saudi Ministry of Higher Education, represented by the Saudi Cultural Bureau in Canada. So, my great thanks to all who made this work successful and supported me to accomplish it.

I also would like to thank everyone in Umm Al-Qura University, and in particular, the Office of Scholarships and University Relations, headed by Dr. Mohammed Basalamah. They did all efforts to make my scholarship smooth and successful. All my thanks to the members of the Custodian of the Two Holy Mosques Institute for Hajj Research, headed by Dr. Osama Al-Barr, who were encouraging and supporting to a great extent.

Yet further, I would like to thank everyone in the University of Victoria, and in particular, the Department of Electrical and Computer Engineering, for their support and guidance until I achieved this work.

Finally, I would like to thank all my friends, Mohammed Fayed, Ahmed Awad, Khalid Khayyat, Dr. Hamdi Sheibani, Yousif Hamad, Mohammed Yaseen, Newaz Rafiq, Mowafaq Househ, Mohammed Wadie, Saeed Zakaria, and others. Without their help, support, friendship, and encouragement, my progress would not be easy.

Dedication

To my mother: The source of affection.

To my wife: The source of love.

To my children: The source of fun.

Chapter 1

Introduction

This chapter is an introduction to the dissertation. It is organized as follows. A brief background on IPSec, authentication, and hash functions is given in § 1.1. In § 1.2, the motivation of the proposed work is discussed. The methodology used in this dissertation is discussed in § 1.3 and the research objectives are given in § 1.4. The road map of the dissertation is given in § 1.5.

1.1 Background

The Internet Protocol (IP) is the standard protocol used for communication in the network layer (or Internet). Because security aspects were not originally considered in the requirements of IP, a number of security threats may threaten an entity receiving an IP packet [1]. These security threats include [2, 3]:

- **Masquerade:** An entity represents itself and pretends to be another.
- **Eavesdropping:** A packet may be spied on by an eavesdropper during communication.
- **Data alteration:** A packet may be modified by an unauthorized entity before it is received at its destination.
- **Replay attacks:** A packet may be resent by a malicious attacker.

In order to provide security to the IP layer and to eliminate the above mentioned security problems, IPSec has been developed by the Internet Engineering Task Force (IETF) [3]. The main security services provided by IPSec include [2]:

- **Confidentiality:** It should not be possible for an unauthorized entity to read the content of a packet during communication. Confidentiality protects against eavesdropping.
- **Data integrity:** The content of a packet should not be altered or modified without detection by the receiver. This protects against data alteration.
- **Message authentication:** The source address of a packet should be of the correct sender and the destination address should not be modified without detection by the receiver. Message authentication is also called *data origin authentication*.
- **Replay protection:** A packet should not be replayed at a later time without detection by the receiver.
- **Key management:** There has to be a way to negotiate the keys required for the cryptographic operations used in the above security services. Key management is also used for entity authentication.

Figure 1.1 shows the main security services provided by IPSec protocols. AH (Authentication Header) is mainly designed for authentication. ESP (Encapsulated Security Payload), on the other hand, provides authentication along with confidentiality. IKE (Internet Key Exchange) is used for key exchange methods. Authentication is an important step to ensure secure sharing and exchanging of keys. Replay protection is provided by the communication mechanism used in IPSec, which uses sequence numbers [2]. It is clear from the figure that authentication plays a very important role for IPSec protocols.

The term *authentication*, when used alone, includes a number of security objectives, such as message authentication, entity authentication, and key authentication. These types of authentication are used to make sure that communication between entities is secure and that no unauthorized entity can compromise this communication. In addition, message au-

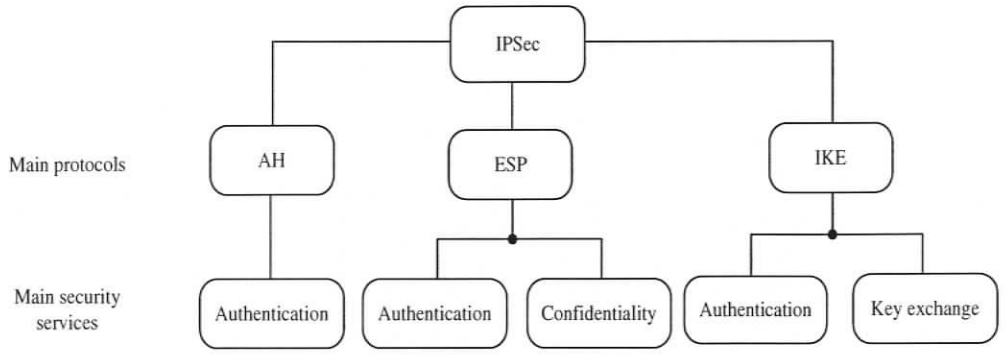

Figure 1.1. *Main security services provided by IPSec protocols.*

thentication implicitly provides data integrity [4]. Figure 1.2 [4] shows that hash functions are used extensively as a building block for authentication and integrity techniques.

More details about IPSec, authentication, and hash functions will be presented in Chapter 2.

1.2 Motivation

As we mentioned in § 1.1, authentication has many applications and plays a very important role for IPSec. Hash functions are the main techniques used for different types of authentication. In particular, HMAC (Keyed-Hash Message Authentication Code) and the three hash functions MD5, SHA-1, and RIPEMD-160 are used extensively by IPSec as shown in Figure 1.3. ESP and AH use HMAC-MD5 (RFC 2403 [5]), HMAC-SHA-1 (RFC 2404 [6]), and HMAC-RIPEMD-160 (RFC 2856 [7]), interchangeably. IKE also uses hash functions, and as stated in the IKE RFC, the selected hash function must support both native and HMAC modes [8]. SHA-1 and MD5 are mandatory for IKE and RIPEMD-160 could be selected optionally. In addition, hash functions may be used for other security functions, such as digital signature. For example, the signature algorithms DSA (Digital Signature Algorithm) and RSA use SHA-1 and MD5 in some of their variants [4, 9, 10].

Therefore, it is important to have these hash functions available for any implementation

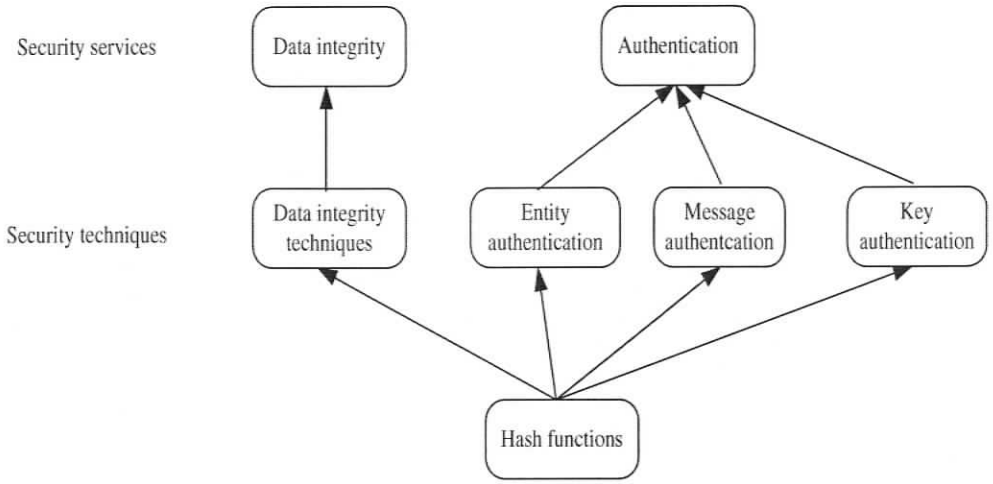

Figure 1.2. *The use of hash functions for authentication and data integrity services.*

of IPSec. However, in hardware implementations of IPSec, it could be costly to have one module for each of these algorithms on the same core. Thus, it is prudent to design a reconfigurable unit that could be used as required for these different security applications. Since MD5, SHA-1, and RIPEMD-160 are based on the same algorithm, MD4, they have a number of similarities that may be exploited to design a unified engine that can perform any one of them.

In this work, we show a design of a unified engine that exploits the similarities among the three hash functions. The integration of the three hash functions on one unified engine is more efficient in area than having one module for each one of them. Furthermore, the similarities between the three hash functions and the parallelism available in the hash algorithms are used to increase throughput and decrease delay of the unified engine. In addition, we integrate this engine to an HMAC unit such that the hash functions can be used alone or with HMAC. This integrated, reconfigurable unit can be used for IPSec or any other application that uses hash functions, such as digital signature.

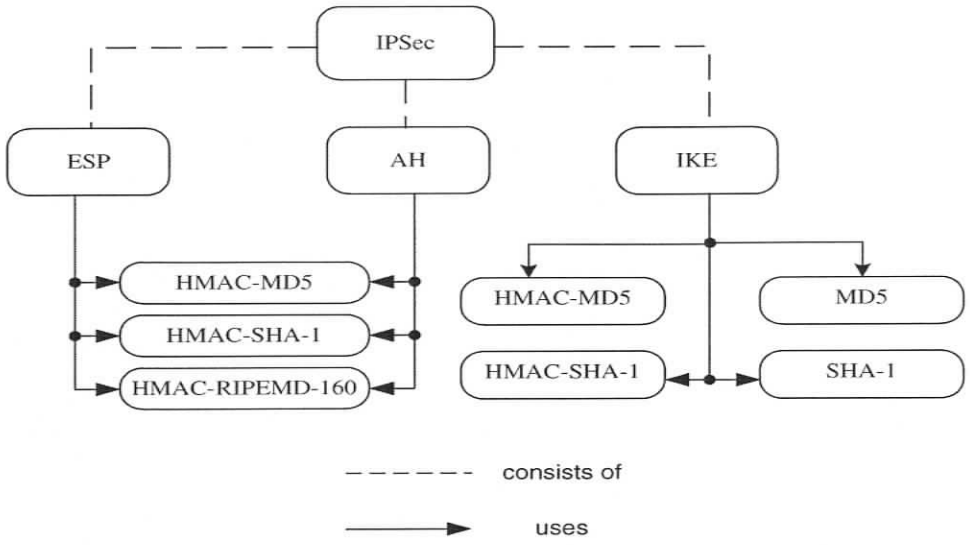

Figure 1.3. *The use of HMAC, MD5, SHA-1, and RIPEMD-160 for IPSec.*

1.3 Methodology

At present, there are different ways to design embedded systems. At one extreme, a project is coded as a software program to run on a general purpose processor (GPP). This method provides the highest level of flexibility and ease of change to accommodate new standards by simply changing the software. However, this option would be poor in performance compared to the other design options. On the other extreme, a target application is implemented using dedicated hardware (or ASIC). While ASICs provide the highest performance, it is difficult to modify the implemented architecture to accommodate new standards since this would require implementation of a new architecture [11].

Between these two extremes, programmable processors and reconfigurable hardware exist to fill the gap between performance and flexibility. Programmable processors, such as ASIPs (Application Specific Instruction set Processors), are designed for specific applications, unlike GPPs which are designed for general purposes [12]. In general, programmable processors are more flexible due to the ease with which software running on them could

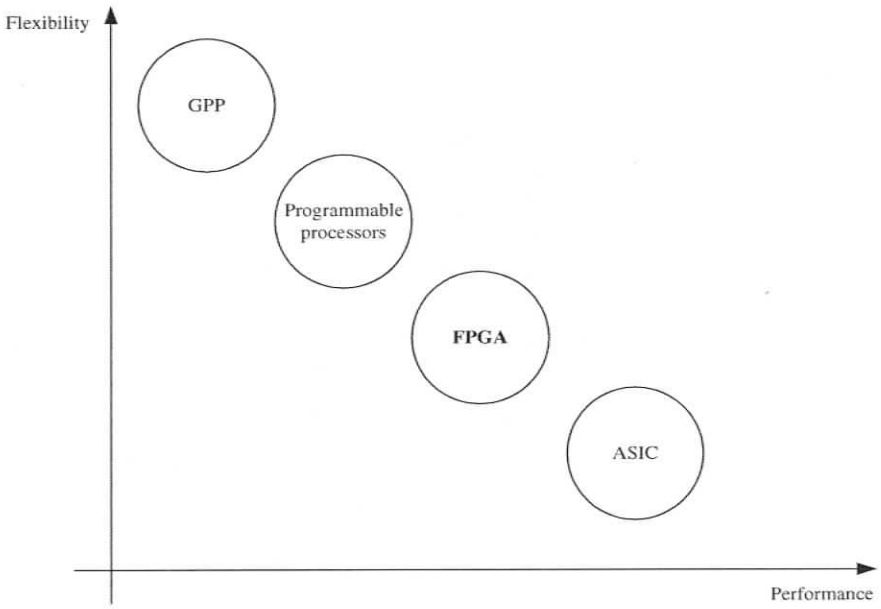

Figure 1.4. *Comparing different design options in terms of performance and flexibility.*

be changed or updated, whereas reconfigurable hardware is better in terms of performance because applications are implemented with dedicated hardware [13]. The common technology used for reconfigurable hardware is Field Programmable Gate Arrays (FPGAs) [11].

Figure 1.4 depicts the above design spectrum. It locates the above described design options and shows their relative performance and flexibility.

Typically, designs targeting GPPs and programmable processors use high level languages, such as C, to implement a target application. High level languages provide the highest level of ease to modify the design with new software updates. On the other hand, the typical way to design ASICs and FPGAs is to use hardware description languages (HDLs), such as VHDL or Verilog, which require knowledge of target devices in order to achieve optimized results [14, 15].

Due to limitations in terms of performance from software-only implementations of embedded systems, most recent efforts focus on hardware implementations. The two most common options for hardware designs are ASICs and FPGAs. In a survey conducted by

Celoxica [16], new design trends are moving towards using FPGAs instead of ASICs for hardware implementations. This is mainly because of the hardware reconfigurability and the low NRE (Non Recurring Engineering) and EDA (Electronic Design Automation) costs provided by FPGA implementations. Although ASICs are still better in terms of performance and unit price, FPGAs are also showing improvement in these two aspects.

A typical embedded systems design methodology prototypes a target design in a high level language, such as C, and then translates it manually into an HDL code. This process is time consuming and error-prone [17]. Furthermore, for efficient designs, the details of target platforms should be known in advance when using HDLs. In other words, HDLs are lower level than high level languages, which reduces the flexibility of a prototype.

In order to narrow the gap between flexibility and performance and reduce the risk of translating a high level prototype into HDLs, we follow an emerging design approach: we use the Handel-C language [18, 19]. Although Handel-C is a high level language, it has the capability to be mapped directly to FPGA. In addition, it has constructs that improve the performance of the design, such as pipelining and parallelism. Moreover, because it is not device-specific, the design can be synthesized to different FPGA chips, allowing more flexibility and speed upgrading.

The methodology we follow in this dissertation allows for a high level of flexibility from two viewpoints, the language level of abstraction and the hardware reconfiguration. Handel-C is like any high level language that can be easily changed for new updates and standards as well as it allows for design space exploration. At the same time, FPGAs are reconfigurable and can be easily reprogrammed for new updates and standards. In addition to narrowing the gap between performance and flexibility, this approach provides fast time-to-market (TTM).

1.4 Research Objectives

The main dissertation objectives can be summarized as follows:

- To design a unified hash unit that implements MD5, SHA-1, and RIPEMD-160 hash functions.
- To design an HMAC unit that implements the HMAC algorithm.
- To integrate the HMAC unit to the unified hash unit to be able to implement any one of MD5, SHA-1, RIPEMD-160, HMAC-MD5, HMAC-SHA-1, and HMAC-RIPEMD-160.
- To implement the designed unit on FPGA platforms using the high level language Handel-C, which enables direct mapping to FPGA chips.
- To enhance the performance of the designed unit by proposing a key reuse technique and applying speedup techniques, such as parallelism and pipelining.
- To compare different HMAC implementations with respect to key size and reuse.
- b. To explore the design space of the proposed HMAC-hash unit by trying different design factors in order to get the most optimal result.
- To analyze the performance of the designed unit and compare achieved results with results reported in previous work.

1.5 Dissertation Road Map

The organization of the dissertation is as follows. Chapter 2 gives background material about IPSec, authentication, and hash functions. In particular, details of HMAC, MD5, SHA-1, and RIPEMD-160 are given.

Chapter 3, provides a literature review on previous work related to the scope of this dissertation. First, a brief review is given on general implementations of IPSec. Then, we review in more details software, ASIC, and FPGA implementations of the hash functions MD5, SHA-1, and RIPEMD-160, whether implementing one hash function or more. After that, a review on implementations of the HMAC algorithm is given, discussing which hash functions are integrated with it.

Chapter 4 discusses the design of the proposed HMAC-hash unit. First, we discuss the unified algorithm used for the three hash functions, MD5, SHA-1, and RIPEMD-160. Then, we discuss how this unified algorithm was mapped onto a hash unit that was integrated to an HMAC unit to form the proposed HMAC-hash unit. The chapter then discusses how we improved the design using some design enhancements, such as parallelism, pipelining, and the proposed key reuse mechanism.

In Chapter 5, the experimental setup is discussed. First, we briefly describe the tools used to model, implement, and analyze the performance of the proposed HMAC-hash unit. Then, we discuss the test strategy we followed to test the designed unit. The experimental steps and design flow are then discussed in details. After that, we discuss the design space exploration applied to design the HMAC-hash unit, and give some examples of using Handel-C to design it.

Chapter 6 discusses the performance analysis of the designed unit and compares it to previous work. First, we analyze the performance of different implementations with respect to key size and key reuse. The performance analysis considers four different factors: area, delay, throughput, and power consumption. Then, we compare the achieved results of this work to results reported in previous work, both implementing hash functions alone or integrating HMAC to some hash functions.

Finally, Chapter 7 concludes this dissertation and gives some guidelines on possible future directions.

Chapter 2

Background Material

This chapter provides background material. Since the main target of the proposed HMAC-hash unit is to be used for IPSec authentication, IPSec is discussed in § 2.1 and authentication is discussed in § 2.2. In § 2.3, hash functions are introduced, and then HMAC, MD5, SHA-1, and RIPEMD-160 are explained in details. As the proposed unit is reconfigurable and the target platform of the design is a reconfigurable hardware, a brief discussion on reconfigurability is given in § 2.4.

2.1 IPSec

IPSec was developed by IETF in order to provide security on the IP layer [3]. IPSec requires that all participating network equipment be consistent. This is achieved using a *Security Association (SA)*, which is a state stored at each end point of a secure connection. One SA is required for each direction of a secure connection. Therefore, for bidirectional communications, two SA's are needed. An SA can be established between two hosts, between a host and a gateway, or between two gateways [2].

An SA includes all information needed for a secure communication, such as the security services to be provided for a specific communication, the cryptographic algorithms to be used, and key lifetimes and how often they should be changed [3].

IPSec communication can operate in one of two modes [2]:

1. **Transport mode:** This mode protects only layers above the IP layer. It can only

be used between end points; i.e., an intermediate node of a secure communication cannot use transport mode.

2. **Tunnel mode:** This mode protects the IP layer as well as above layers. This is done by encapsulating the whole IP packet inside an IPSec datagram.

IPSec is a set of security protocols. The main concepts of IPSec are described in RFC 2401 [20]. The main security protocols provided by IPSec are [2]:

1. **Authentication Header (AH):** This protocol provides data origin authentication, data integrity, and anti-replay protection, but does not provide data confidentiality. This protocol is specified in RFC 2402 [21].
2. **Encapsulated Security Payload (ESP):** This protocol provides a combined authentication/encryption facility to add data confidentiality to the security services provided by AH. At least one of the two main security services provided (authentication and confidentiality) must be selected for an SA, and they may also be combined. This protocol is specified in RFC 2406 [22].
3. **Key management protocols:** There are two main key management protocols: Internet Security Association and Key Management Protocol (ISAKMP), which is specified in RFC 2408 [23], and Internet Key Exchange (IKE), which is specified in RFC 2409 [8]. ISAKMP defines a framework for authentication, key exchange, and negotiation of different parameters required for an SA. Based on the framework defined by ISAKMP, IKE organizes the process of exchanging security keys and ensures their privacy.

2.2 Authentication

Authentication is one of the most important cryptographic services used to guarantee secure communication [4]. The term *authentication* is used in a broad sense to cover a number of security objectives. To be more specific, authentication is classified into two main types [4]:

1. **Message authentication:** in which a receiving party is assured that the source of the received data is as claimed. It is also called *data origin authentication*.
2. **Entity authentication:** which provides assurance to a communicating entity that the second entity is as claimed and that it was active at the time of sending its identity.

Message authentication implicitly provides *data integrity*, which is the assurance that data received has not been modified by an unauthorized entity during transmission. If the received data has been modified, the originator of the message is no longer the correct one, and vice versa. Techniques used to provide message authentication include message authentication codes (MACs), digital signature, and appending a secret authenticator to encrypted messages. If digital signature is used for message authentication, another security service is provided, which is non-repudiation. Non-repudiation is a service that prevents an entity from denying previous actions or commitments.

Usually, message authentication does not provide uniqueness and timeliness assurance. If uniqueness and timeliness guarantees are added to message authentication, it is called *transaction authentication*. There are a number of techniques for transaction authentication, such as timestamps and sequence numbers.

Entity authentication can be achieved by using a number of techniques, such as personal identification numbers (PINs) and passwords, challenge-response mechanisms, and customized identification protocol.

Key authentication is used when a key is involved in message or entity authentication. Key authentication assures a communicating entity that a secret key cannot be accessed by any other entity except the identified one. This is called *implicit key authentication*. If, in addition to this, a communicating entity is assured that the other entity possess the secret key, *explicit key authentication* is guaranteed. Usually, key authentication is combined with entity authentication when designing key establishment protocols.

Figure 2.1 shows the main building blocks of different authentication and integrity techniques [4]. It is clear from the figure the important role of hash functions for authentication and other security objectives, such as data integrity and non-repudiation. In the following

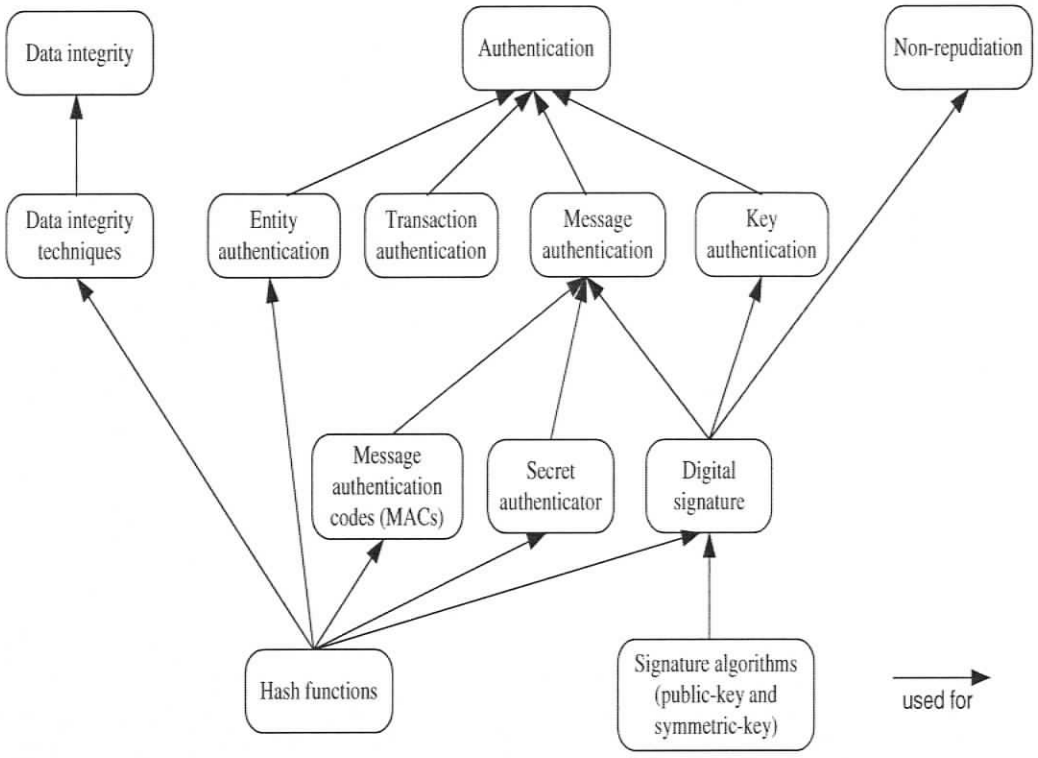

Figure 2.1. *Main building blocks of authentication and data integrity services.*

section, hash functions are discussed.

2.3 Hash Functions

Hash functions are important security primitives used in modern cryptography. In its simplest form, a hash function takes an input message of arbitrary length and produces a fixed size output. The output of a hash function is called *hash code, hash result, hash value, message digest,* or simply *hash* [4]. Any hash function h has to have at least the following two properties:

1. **Compression:** To map an input message M of arbitrary finite length to a fixed length output $h(M)$.

2. **Ease of computation:** It should be easy to compute the hash value $h(M)$ given the input message M.

One way hash functions (OWHF) add an additional property, which is called *preimage resistance*. Preimage resistance has the following characteristics:

- If h and $h(M)$ are known, it is computationally infeasible to find M. In other words, it is computationally infeasible to find an input message that hashes to a given hash value.
- Given M and h, it is computationally infeasible to find another message M' such that $h(M) = h(M')$.

A stronger type of hash functions is called collision resistant hash function (CRHF), or collision free hash function (CFHF) [4, 24]. A CRHF adds the following property to the above characteristics:

- Given a hash function h, it is infeasible to find two different messages M and M' such that $h(M) = h(M')$.

OWHFs are also called weak one way hash functions, whereas CRHFs are called strong one way hash functions.

There are a number of types of hash functions. The most popular ones are [4]:

1. Hash functions based on block ciphers.
2. Hash functions based on modular arithmetic.
3. Customized (or dedicated) hash functions, which are functions designed from scratch for the purpose of hashing.

Other types of hash functions include hash functions based on cellular automaton, on Knapsack problem, and on algebraic matrices [24].

Hash functions may be used with a secret key, and in that case they are called keyed hash functions, or used without a key, and then they are called un-keyed hash functions. The most popular keyed hash functions are those used for message authentication, and thus

called message authentication codes (MACs). Among the most used MACs is the Keyed-Hash Message Authentication Code (HMAC). The most popular un-keyed hash functions are called modification detection codes (MDCs), and among the most used MDCs are MD5, SHA-1, and RIPEMD-160 [4]. In the following subsections, we introduce these hash functions.

2.3.1 Keyed-Hash Message Authentication Code (HMAC)

HMAC is a shared-key security algorithm that uses hash functions for authentication. Its strength is based on the strength of the underlying hash function. The details of the HMAC algorithm are given in [25, 26].

HMAC gets a message of arbitrary length M and produces a fixed length output MAC. It uses a secret key and an un-keyed hash function to compute the MAC. The main operation of the HMAC algorithm is given by the following equation:

$$MAC(M) = h((K_0 \oplus opad)@(h((K_0 \oplus ipad)@M))) \tag{2.1}$$

This equation and the symbols used in it are explained in the following algorithm:

Algorithm 2.1 *HMAC Algorithm*

Declarations:

- I_b*: Number of bytes of the input to the hash function.*
- O_b*: Number of bytes of the output from the hash function.*
- $ipad$*: The constant byte 0x36 repeated* I_b *times.*
- $opad$*: The constant byte 0x5C repeated* I_b *times.*
- K*: Shared secret key.*
- K_b*: Number of bytes of* K*.*
- K_0*:* K *after resizing to* I_b *bytes.*
- M*: The message to be hashed.*

- MAC: *The output hashed value of* M *using HMAC.*
- $h(M)$: *The hash value resulted from hashing the message* M *using the hash function* h.
- $Trunc_n(V)$: *Truncation of variable* V *to the* n *leftmost bits.*
- V @ $(0x00)_n$: *Padding of* n *bytes of zeros to the right of variable* V.
- $V1$ *to* $V6$ *are temporary variables.*

The algorithm:

1. *Resize the key* K:

$$K_0 = \begin{cases} K@(0x00)_{I_b - K_b}, & \text{if } K_b < I_b \\ K, & \text{if } K_b = I_b \\ h(K)@(0x00)_{I_b - O_b}, & \text{if } K_b > I_b \end{cases} \tag{2.2}$$

2. $V1 = K_0 \oplus ipad$
3. $V2 = V1$ @ M
4. $V3 = h(V2)$
5. $V4 = K_0 \oplus opad$
6. $V5 = V4$ @ $V3$
7. $V6 = h(V5)$
8. $MAC = Trunc_t(V6) \qquad \frac{O_b}{2} \leq t \leq O_b$

2.3.2 MD5

The MD5 algorithm is a hash function used to hash a message M of size MS bits, where $0 \leq MS < 2^{64}$. MD5 consists of the following steps [4, 27]:

1. **Initialization**: In this step, the constants required for MD5 are defined. These constants include:

- An additive 32-bit constant, $y_m[j]$, which is equal to the first 32 bits of the absolute binary value $abs(sin(j+1))$, where $0 \leq j \leq 63$, and j is in radians. This can be expanded as follows (in hexadecimal) [9]:

  $y_m[0..3]$ = [0xd76aa478, 0xe8c7b756, 0x242070db, 0xc1bdceee]

  $y_m[4..7]$ = [0xf57c0faf, 0x4787c62a, 0xa8304613, 0xfd469501]

  $y_m[8..11]$ = [0x698098d8, 0x8b44f7af, 0xffff5bb1, 0x895cd7be]

  $y_m[12..15]$ = [0x6b901122, 0xfd987193, 0xa679438e, 0x49b40821]

  $y_m[16..19]$ = [0xf61e2562, 0xc040b340, 0x265e5a51, 0xe9b6c7aa]

  $y_m[20..23]$ = [0xd62f105d, 0x02441453, 0xd8a1e681, 0xe7d3fbc8]

  $y_m[24..27]$ = [0x21e1cde6, 0xc33707d6, 0xf4d50d87, 0x455a14ed]

  $y_m[28..31]$ = [0xa9e3e905, 0xfcefa3f8, 0x676f02d9, 0x8d2a4c8a]

  $y_m[32..35]$ = [0xfffa3942, 0x8771f681, 0x6d9d6122, 0xfde5380c]

  $y_m[36..39]$ = [0xa4beea44, 0x4bdecfa9, 0xf6bb4b60, 0xbebfbc70]

  $y_m[40..43]$ = [0x289b7ec6, 0xeaa127fa, 0xd4ef3085, 0x04881d05]

  $y_m[44..47]$ = [0xd9d4d039, 0xe6db99e5, 0x1fa27cf8, 0xc4ac5665]

  $y_m[48..51]$ = [0xf4292244, 0x432aff97, 0xab9423a7, 0xfc93a039]

  $y_m[52..55]$ = [0x655b59c3, 0x8f0ccc92, 0xffeff47d, 0x85845dd1]

  $y_m[56..59]$ = [0x6fa87e4f, 0xfe2ce6e0, 0xa3014314, 0x4e0811a1]

  $y_m[60..63]$ = [0xf7537e82, 0xbd3af235, 0x2ad7d2bb, 0xeb86d391]

- An access order, $z_m[j]$, which is defined as follows:

  $z_m[0..15]$ = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]

  $z_m[16..31]$ = [1, 6, 11, 0, 5, 10, 15, 4, 9, 14, 3, 8, 13, 2, 7, 12]

  $z_m[32..47]$ = [5, 8, 11, 14, 1, 4, 7, 10, 13, 0, 3, 6, 9, 12, 15, 2]

  $z_m[48..63]$ = [0, 7, 14, 5, 12, 3, 10, 1, 8, 15, 6, 13, 4, 11, 2, 9]

- The number of bits for rotate-left operation, $s_m[j]$, which is defined as follows:

  $s_m[0..15]$ = [7, 12, 17, 22, 7, 12, 17, 22, 7, 12, 17, 22, 7, 12, 17, 22]

  $s_m[16..31]$ = [5, 9, 14, 20, 5, 9, 14, 20, 5, 9, 14, 20, 5, 9, 14, 20]

  $s_m[32..47]$ = [4, 11, 16, 23, 4, 11, 16, 23, 4, 11, 16, 23, 4, 11, 16, 23]

$$s_m[48..63] = [6, 10, 15, 21, 6, 10, 15, 21, 6, 10, 15, 21, 6, 10, 15, 21]$$

- Initial chaining values (IVs), denoted by Ci, which are set to the following hexadecimal values:

$$C1 = 0x67452301$$
$$C2 = 0xefcdab89$$
$$C3 = 0x98badcfe$$
$$C4 = 0x10325476$$

MD5 uses four chaining variables, $H1$ to $H4$, each is 32 bits. These chaining variables are updated after processing each message block M^i. The IVs defined above are used to initialize these chaining variables as follows:

$$H1 = C1$$
$$H2 = C2$$
$$H3 = C3$$
$$H4 = C4$$

2. **Preprocessing**: MD5 works on 512-bit blocks, one block at a time. Therefore, the message to be hashed has to be of a length divisible by 512 and then segmented into 512-bit blocks. This is done through the following sub-steps:
   (a) **Padding**: The goal of padding is to make the length of the message a multiple of 512. This is done by appending a single bit of value '1' followed by l zero bits, where l is the smallest non-negative solution to the equation $MS + 1 + l = 448 \bmod 512$. Then, 64 bits representing the message length MS are appended. The representation of the padded message must be in little-endian format, where each 32-bit word consists of four bytes. The least significant byte comes first and the most significant byte comes last.
   (b) **Parsing**: Divide the padded message into N 512-bit blocks M^1 to M^N.
3. **Processing**: The message blocks M^1 to M^N are processed in order. In each step i,

the message block M^i is segmented into 16 32-bit variables X_0 to X_{15} and processed as follows:

(a) Working variables initialization:

Initialize the four 32-bit working variables A, B, C, and D to the current values of the chaining variables such that:

$$A = H1$$
$$B = H2$$
$$C = H3$$
$$D = H4$$

(b) Compression step:

For j = 0 to 63 Do

$$T = A + f_j(B, C, D) + X_{z_m[j]} + y_m[j]$$
$$A = D$$
$$D = C$$
$$C = B$$
$$B = B + ROTL^{s_m[j]}(T)$$

where $f_j(u, v, w)$ is a logical function applied to the 32-bit variables u, v, and w in step j. MD5 has 4 rounds, each consists of 16 compression steps and each round uses the same logical function f_j in all its steps as follows:

$$f_j(u, v, w) = \begin{cases} (u \,\&\, v) \mid (\overline{u} \,\&\, w), & 0 \leq j \leq 15 \\ (u \,\&\, w) \mid (v \,\&\, \overline{w}), & 16 \leq j \leq 31 \\ u \oplus v \oplus w, & 32 \leq j \leq 47 \\ v \oplus (u \mid \overline{w}), & 48 \leq j \leq 63 \end{cases}$$

(c) Chaining variables update:

Update the chaining variables as follows:

$$H1 = A + H1$$
$$H2 = B + H2$$
$$H3 = C + H3$$
$$H4 = D + H4$$

4. **Completion**: The message digest produced by MD5 is a 128-bit variable, denoted by $Hash$, and computed by concatenating the chaining variables obtained after processing the last message block M^N as follows:

$$Hash = H1 \ @ \ H2 \ @ \ H3 \ @ \ H4$$

The representation of the message digest must be in little-endian format.

2.3.3 SHA-1

The SHA-1 algorithm is a hash function used to hash a message M of size MS bits, where $0 \leq MS < 2^{64}$. SHA-1 consists of the following steps [4, 28, 29]:

1. **Initialization**: In this step, the constants required for SHA-1 are defined. These constants include:
   - An additive 32-bit constant, $y_s[j]$, whose hexadecimal value is given as:

$$y_s[j] = \begin{cases} 0x5a827999, & 0 \leq j \leq 19 \\ 0x6ed9eba1, & 20 \leq j \leq 39 \\ 0x8f1bbcdc, & 40 \leq j \leq 59 \\ 0xca62c1d6, & 60 \leq j \leq 79 \end{cases}$$

   - Initial chaining values (IVs), denoted by Ci, which are set to the following hexadecimal values:

$C1$ = 0x67452301

$C2$ = 0xefcdab89

$C3$ = 0x98badcfe

$C4$ = 0x10325476

$C5$ = 0xc3d2e1f0

SHA-1 uses five chaining variables $H1$ to $H5$, each is 32 bits. These chaining variables are updated after processing each message block M^i. The IVs defined above are used to initialize these chaining variables as follows:

$H1 = C1$

$H2 = C2$

$H3 = C3$

$H4 = C4$

$H5 = C5$

2. **Preprocessing**: SHA-1 works on 512-bit blocks, one block at a time. Therefore, the message to be hashed has to be of a length divisible by 512 and then segmented into 512-bit blocks. This is done through the following sub-steps:

   (a) **Padding**: The goal of padding is to make the length of the message a multiple of 512. This is done by appending a single bit of value '1' followed by l zero bits, where l is the smallest non-negative solution to the equation $MS + 1 + l = 448 \bmod 512$. Then, 64 bits representing the message length MS are appended.

   (b) **Parsing**: Divide the padded message into N 512-bit blocks M^1 to M^N.

3. **Processing**: The message blocks M^1 to M^N are processed in order. In each step i, the message block M^i is segmented into 16 32-bit variables X_0 to X_{15} and processed as follows:

   1. **Word expansion:**

      The 16 32-bit variables X_0 to X_{15} are expanded into 80 32-bit words X_0 to X_{79} as follows:

$$X_j = \begin{cases} M_j^i, & 0 \leq j \leq 15 \\ ROTL^1(X_{j-3} \oplus X_{j-8} \oplus X_{j-14} \oplus X_{j-16}), & 16 \leq j \leq 79 \end{cases}$$

where M_0^i is the most significant 32 bits of message block M^i.

2. **Working variables initialization:**

   Initialize the 32-bit working variables A, B, C, D, and E to the current values of the chaining variables:

$$A = H1$$
$$B = H2$$
$$C = H3$$
$$D = H4$$
$$E = H5$$

3. **Compression step:**

   For j = 0 to 79 Do

$$T = ROTL^5(A) + f_j(B, C, D) + E + y_s[j] + X_j$$
$$E = D$$
$$D = C$$
$$C = ROTL^{30}(B)$$
$$B = A$$
$$A = T$$

where $f_j(u, v, w)$ is a logical function applied to the 32-bit variables u, v, and w in step j. SHA-1 has 4 rounds, each consists of 20 compression steps and each round uses the same logical function f_j in all its steps as follows:

$$f_j(u,v,w) = \begin{cases} (u \,\&\, v) \mid (\overline{u} \,\&\, w), & 0 \leq j \leq 19 \\ u \oplus v \oplus w, & 20 \leq j \leq 39 \\ (u \,\&\, v) \mid (u \,\&\, w) \mid (v \,\&\, w), & 40 \leq j \leq 59 \\ u \oplus v \oplus w, & 60 \leq j \leq 79 \end{cases}$$

4. **Chaining variables update:**

   Update the chaining variables such that:

$$H1 = A + H1$$
$$H2 = B + H2$$
$$H3 = C + H3$$
$$H4 = D + H4$$
$$H5 = E + H5$$

5. **Completion**: The message digest produced by SHA-1 is a 160-bit variable, denoted by $Hash$, and computed by concatenating the chaining variables obtained after processing the last message block M^N as follows:

$$Hash = H1 \ @ \ H2 \ @ \ H3 \ @ \ H4 \ @ \ H5$$

2.3.4 RIPEMD-160

The RIPEMD-160 algorithm is a hash function used to hash a message M of size MS bits, where $0 \leq MS < 2^{64}$. RIPEMD-160 consists of the following steps [4, 30, 31]:

1. **Initialization**: In this step, the constants required for RIPEMD-160 are defined. These constants include:
   - Additive 32-bit constants, $y_{RL}[j]$ and $y_{RR}[j]$, whose hexadecimal values are:

$$y_{RL}[j] = \begin{cases} 0, & 0 \leq j \leq 15 \\ 0x5a827999, & 16 \leq j \leq 31 \\ 0x6ed9eba1, & 32 \leq j \leq 47 \\ 0x8f1bbcdc, & 48 \leq j \leq 63 \\ 0xa953fd4e, & 64 \leq j \leq 79 \end{cases}$$

$$y_{RR}[j] = \begin{cases} 0x50a28be6, & 0 \leq j \leq 15 \\ 0x5c4dd124, & 16 \leq j \leq 31 \\ 0x6d703ef3, & 32 \leq j \leq 47 \\ 0x7a6d76e9, & 48 \leq j \leq 63 \\ 0, & 64 \leq j \leq 79 \end{cases}$$

- Access order constants, $z_{RL}[j]$ and $z_{RR}[j]$, defined as follows:

  $z_{RL}[0..15]$ = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]

  $z_{RL}[16..31]$ = [7, 4, 13, 1, 10, 6, 15, 3, 12, 0, 9, 5, 2, 14, 11, 8]

  $z_{RL}[32..47]$ = [3, 10, 14, 4, 9, 15, 8, 1, 2, 7, 0, 6, 13, 11, 5, 12]

  $z_{RL}[48..63]$ = [1, 9, 11, 10, 0, 8, 12, 4, 13, 3, 7, 15, 14, 5, 6, 2]

  $z_{RL}[64..79]$ = [4, 0, 5, 9, 7, 12, 2, 10, 14, 1, 3, 8, 11, 6, 15, 13]

  $z_{RR}[0..15]$ = [5, 14, 7, 0, 9, 2, 11, 4, 13, 6, 15, 8, 1, 10, 3,12]

  $z_{RR}[16..31]$ = [6, 11, 3, 7, 0, 13, 5, 10, 14, 15, 8, 12, 4, 9, 1, 2]

  $z_{RR}[32..47]$ = [15, 5, 1, 3, 7, 14, 6, 9, 11, 8, 12, 2, 10, 0, 4, 13]

  $z_{RR}[48..63]$ = [8, 6, 4, 1, 3, 11, 15, 0, 5, 12, 2, 13, 9, 7, 10, 14]

  $z_{RR}[64..79]$ = [12, 15, 10, 4, 1, 5, 8, 7, 6, 2, 13, 14, 0, 3, 9, 11]

- The number of bits for rotate-left operation, $s_{RL}[j]$ and $s_{RR}[j]$, defined as follows:

  $s_{RL}[0..15]$ = [11, 14, 15, 12, 5, 8, 7, 9, 11, 13, 14, 15, 6, 7, 9, 8]

$s_{RL}[16..31]$ = [7, 6, 8, 13, 11, 9, 7, 15, 7, 12, 15, 9, 11, 7, 13, 12]

$s_{RL}[32..47]$ = [11, 13, 6, 7, 14, 9, 13, 15, 14, 8, 13, 6, 5, 12, 7, 5]

$s_{RL}[48..63]$ = [11, 12, 14, 15, 14, 15, 9, 8, 9, 14, 5, 6, 8, 6, 5, 12]

$s_{RL}[64..79]$ = [9, 15, 5, 11, 6, 8, 13, 12, 5, 12, 13, 14, 11, 8, 5, 6]

$s_{RR}[0..15]$ = [8, 9, 9, 11, 13, 15, 15, 5, 7, 7, 8, 11, 14, 14, 12, 6]

$s_{RR}[16..31]$ = [9, 13, 15, 7, 12, 8, 9, 11, 7, 7, 12, 7, 6, 15, 13, 11]

$s_{RR}[32..47]$ = [9, 7, 15, 11, 8, 6, 6, 14, 12, 13, 5, 14, 13, 13, 7, 5]

$s_{RR}[48..63]$ = [15, 5, 8, 11, 14, 14, 6, 14, 6, 9, 12, 9, 12, 5, 15, 8]

$s_{RR}[64..79]$ = [8, 5, 12, 9, 12, 5, 14, 6, 8, 13, 6, 5, 15, 13, 11, 11]

- Initial chaining values (IVs), denoted by Ci, which are set to the following hexadecimal values:

$$C1 = 0x67452301$$
$$C2 = 0xefcdab89$$
$$C3 = 0x98badcfe$$
$$C4 = 0x10325476$$
$$C5 = 0xc3d2e1f0$$

RIPEMD-160 uses five chaining variables $H1$ to $H5$, each is 32 bits. These chaining variables are updated after processing each message block M^i. The IVs defined above are used to initialize these chaining variables as follows:

$$H1 = C1$$
$$H2 = C2$$
$$H3 = C3$$
$$H4 = C4$$
$$H5 = C5$$

2. **Preprocessing**: RIPEMD-160 works on 512-bit blocks, one block at a time. Therefore, the message to be hashed has to be of a length divisible by 512 and then seg-

mented into 512-bit blocks. This is done through the following sub-steps:

(a) **Padding**: The goal of padding is to make the length of the message a multiple of 512. This is done by appending a single bit of value '1' followed by l zero bits, where l is the smallest non-negative solution to the equation $MS + 1 + l = 448 \bmod 512$. Then, 64 bits representing the message length MS are appended. The representation of the padded message must be in little-endian format, where each 32-bit word consists of four bytes. The least significant byte comes first and the most significant byte comes last.

(b) **Parsing**: Divide the padded message into N 512-bit blocks M^1 to M^N.

3. **Processing**: The message blocks M^1 to M^N are processed in order. In each step i, the message block M^i is divided into 16 32-bit variables X_0 to X_{15}. Then, two parallel lines of processing are performed, *left line* and *right line*. This is done as follows:

   1. **Working variables initialization:**

      Initialize five 32-bit working variables for each processing line, AL to EL for the left line and AR to ER for the right line. These working variables are initialized as follows:

$$AL = AR = H1$$
$$BL = BR = H2$$
$$CL = CR = H3$$
$$DL = DR = H4$$
$$EL = ER = H5$$

   2. **Compression step:**

      For $j = 0$ to 79 Do

      (For the left processing line):

$$TL = AL + FL_j(BL, CL, DL) + X_{z_{RL}[j]} + y_{RL}[j]$$

$$AL = EL$$
$$BL = EL + ROTL^{s_{RL}[j]}(TL)$$
$$CL = BL$$
$$DL = ROTL^{10}(CL)$$
$$EL = DL$$

(For the right processing line):

$$TR = AR + FR_j(BR, CR, DR) + X_{z_{RR}[j]} + y_{RR}[j]$$
$$AR = ER$$
$$BR = ER + ROTL^{s_{RR}[j]}(TR)$$
$$CR = BR$$
$$DR = ROTL^{10}(CR)$$
$$ER = DR$$

where $FL_j(u, v, w)$ and $FR_j(u, v, w)$ are logical functions applied to the 32-bit variables u, v, and w in step j. Each processing line of RIPEMD-160 has 5 rounds, each consists of 16 compression steps and each round uses the same logical function in all its steps as follows:

(For the left processing line):

$$FL_j(u, v, w) = \begin{cases} u \oplus v \oplus w, & 0 \leq j \leq 15 \\ (u \,\&\, v) \mid (\overline{u} \,\&\, w), & 16 \leq j \leq 31 \\ (u \mid \overline{v}) \oplus w, & 32 \leq j \leq 47 \\ (u \,\&\, w) \mid (v \,\&\, \overline{w}), & 48 \leq j \leq 63 \\ u \oplus (v \mid \overline{w}), & 64 \leq j \leq 79 \end{cases}$$

(For the right processing line):

$$FR_j(u,v,w) = \begin{cases} u \oplus (v \mid \overline{w}), & 0 \leq j \leq 15 \\ (u \,\&\, w) \mid (v \,\&\, \overline{w}), & 16 \leq j \leq 31 \\ (u \mid \overline{v}) \oplus w, & 32 \leq j \leq 47 \\ (u \,\&\, v) \mid (\overline{u} \,\&\, w), & 48 \leq j \leq 63 \\ u \oplus v \oplus w, & 64 \leq j \leq 79 \end{cases}$$

3. **Chaining variables update:**

   Update the chaining variables as follows:

$$H1 = H2 + CL + DR$$
$$H2 = H3 + DL + ER$$
$$H3 = H4 + EL + AR$$
$$H4 = H5 + AL + BR$$
$$H5 = H1 + BL + CR$$

4. **Completion**: The message digest produced by RIPEMD-160 is a 160-bit variable, denoted by $Hash$, and computed by concatenating the chaining variables obtained after processing the last message block M^N as follows:

$$Hash = H1 \ @ \ H2 \ @ \ H3 \ @ \ H4 \ @ \ H5$$

   The representation of the message digest must be in little-endian format.

2.4 Reconfigurability

Reconfigurable computing is used in the literature to refer to systems that have a level of hardware programmability [11]. The logic functionality and interconnect of a reconfigurable device is customizable by user-defined programs and can be reprogrammed as required [13]. Reconfigurable hardware is intended to narrow the gap between ASICs and

general purpose processors. ASICs provide the highest level of performance but are inflexible since a new ASIC has to be designed for new applications. General purpose processors, on the other hand, are the most flexible and can be easily programmed but are poor in performance because they use software implementations [11, 13].

The common technology used for reconfigurable hardware is Field Programmable Gate Arrays (FPGAs) [11]. An FPGA consists of three main parts, logic blocks, input/output blocks (IOBs), and programmable interconnect [32]. Logic blocks are used to implement a target application, IOBs are used either as inputs pad or output pads, and the interconnect is used for routing [32]. Usually, a logic block consists of lookup tables (LUTs), flip flops, carry logic, and programmable multiplexors [32, 33]. In addition, some FPGAs include other resources, such as dedicated memory resources, dedicated arithmetic and logic units (ALUs), and processor cores [33]. The design trend is moving from ASIC-based to FPGA-based, according to [16].

Reconfigurability can be classified into two types [13, 34]:

1. **Static:** A reconfigurable hardware is configured once and then it does not change during execution. If a new application is to be implemented, the reconfigurable device needs to be configured again.
2. **Dynamic:** The logic and/or interconnect of the reconfigurable hardware can be modified during execution using some control signals. There are three types of dynamic reconfigurations [35]:
   (a) **Algorithmic reconfigurations:** The system is reconfigured to implement the same functionality using different computational algorithms that differ in performance, accuracy, power, or resource requirements.
   (b) **Architectural reconfigurations:** Hardware components of the reconfigurable device are modified for different computations.
   (c) **Functional reconfigurations:** Different functions are executed in the same hardware resources.

The design of FPGAs needs assistance of some design tools. The most complex approach is to design the FPGA manually to gain the highest level of performance. On the other hand, fully automatic design tools may be used to allow more flexibility at the expense of performance [11]. The typical approach used in the literature is to use hardware description languages, such as VHDL and Verilog, to design the desired application and then use a synthesis tool to translate it into bit files. This approach comes between the above mentioned two approaches [17].

2.5 Concluding Remarks

In this chapter, background material of the topics relevant to the work of this dissertation has been given. This covers IPSec, authentication, hash functions, and reconfigurability. In particular, we have discussed HMAC, MD5, SHA-1, and RIPEMD-160 in details.

In the following chapter, we provide a literature review on previous work related to the scope of this dissertation.

Chapter 3

Previous Work

In this chapter, a review of previous work related to this dissertation is provided. First, we briefly review work that addressed IPSec implementations in § 3.1. In § 3.2, we review work that implemented one hash function. We focus on MD5, SHA-1, and RIPEMD-160. Work that implemented two and three hash functions is reviewed in § 3.3 and § 3.4, respectively. In § 3.5, we review work that integrated HMAC with some hash functions.

3.1 IPSec Implementations

Most of the work implementing IPSec focused on confidentiality, which is provided using encryption algorithms, and authentication. Some work focused on software implementation of IPSec. Lin et al. [36] implemented IPSec in software in order to evaluate the impact of various IPSec services and algorithms on router performance. For hashing algorithms, they focused on MD5 and SHA-1 and compared the two algorithms for AH and ESP.

Leffler [37] implemented a software version of IPSec (called Fast IPSec) for FreeBSD (which is a free version of UNIX operating system). Fast IPSec runs on a uniprocessor system. The cryptographic algorithms implemented are those specified in the FreeBSD framework, i.e., 3DES (Triple Data Encryption Standard), MD5, and SHA-1. Leffler optimized these algorithms for software implementation in [38].

Barton et al. [39] integrated IPSec with Mobile IPv4 in a software implementation to provide security for wireless communication. They used the FreeS/WAN IPSec software

(a Linux implementation of the IPSec protocols) in their implementation.

Other work focused on hardware implementation of encryption algorithms for IPSec. Cheung et al. [40, 41] implemented an IPSec accelerator for VPN (Virtual Private Network). The focus was on the encryption algorithms DES, 3DES, and IDEA (International Data Encryption Algorithm). VHDL was used to implement the IPSec accelerator on a Xilinx FPGA chip.

Dandalis et al. [42, 43] implemented an adaptive cryptographic engine for IPSec on a Xilinx FPGA chip. They focused on encryption algorithms. In particular, they implemented five symmetric-key algorithms, MARS, RC5, Rijndael, Serpent, and Twofish in an adaptive architecture built on FPGA. These five algorithms were the final candidates for AES (Advanced Encryption Standard) announced by NIST (National Institute of Standards & Technology) in 1999. In 2000, Rijndael was selected to be the AES standard.

Elbirt [44] also focused on implementing the AES candidate algorithms, RC5, Rijndael, Serpent, and Twofish. He used VHDL to implement them on a reconfigurable architecture using a Xilinx FPGA chip. He called his proposed reconfigurable architecture COBRA, which stands for Cryptographic (Optimized for Block Ciphers) Reconfigurable Architecture.

Chodowiec et al. [45] implemented the Rijndael and 3DES algorithms on Xilinx FPGA chips. The goal of their work is to design a cryptographic accelerator for IPSec.

There are a number of work that focused on IPSec authentication. Lee and Kwak [46] designed a cryptographic accelerator that implements MD5 and Elliptic Curve Scalar Multiplier on FPGA for IPSec authentication. Kang et al. [47] implemented MD5, SHA-1, and HAS-160 (The Hash function Algorithm Standard). Some other literature combined authentication and encryption. Ha et al. [48] implemented AES-128/119/256, 3DES (or TDES), HMAC-MD5, and HMAC-SHA-1 on an ASIC accelerator for IPSec. McLoone and McCanny [49] combined HMAC-SHA-1 and Rijndael on an IPSec chip implemented on FPGA. Lu and Lockwood [50] implemented HMAC-SHA-1 and HMAC-MD5 and AES/CBC on FPGA.

The above mentioned literature is explicitly designed for IPSec. There exists other literature that implemented hash functions, whether alone or integrated with HMAC. In the following sections, we review this literature. For the sake of completeness, we also review work on IPSec authentication in more details. We focus on HMAC, MD5, SHA-1, and RIPEMD-160. Although we review some software implementations in each section, our focus is on hardware implementations.

3.2 Implementation of One Hash Function

In this section, we review work that implemented only one hash function, whether alone or combined with some other cryptographic algorithms. We focus on the three hash functions MD5, SHA-1, and RIPEMD-160.

3.2.1 MD5 Implementations

A number of literature focused on software implementation of MD5. Oliva et al. [51] combined MD5 with AES, DES, and 3DES in a programmable processor, called Cryptonite. However, the focus in their work was on AES implementation.

Nahum et al. [52] implemented MD5 and other cryptographic algorithms in software to prove that software implementation is not suitable for high speed networks. To improve the performance of software implementations, they proposed three approaches: the first is to design faster algorithms, the second is to use parallelism, and the third is to add some instructions to the standard instruction sets in order to improve the performance of cryptographic algorithms.

Touch [53, 54] analyzed the performance of software implementation of MD5 and concluded that the performance of MD5 software implementations is not sufficient for high speed networks. He proposed modifying the MD5 algorithm or using alterative hash algorithms to improve performance.

Arnold [55] implemented MD5 on the NAPA architecture (National Semiconductor's

Adaptive Processing Architecture). The NAPA architecture integrates a Fixed Instruction set Processor (FIP), an Adaptive Logic Processor (ALP), and some memory and other circuitry into a single reconfigurable device. Different MD5 implementations on the NAPA architecture were compared.

Most of the recent work focused on hardware implementations of IPSec. Deepakumara et al. [56] compared two implementations of MD5. The first one uses iterative implementation, where an iterative core is designed to implement the compression steps of MD5. The second implementation uses full loop unrolling, where each compression step of MD5 has its own combinational logic. The iterative implementation consumes less area but has higher delay than the second one. They used VHDL to implement the two implementations on a Xilinx FPGA chip.

Lee and Kwak [46] combined MD5 and Elliptic Curve Scalar Multiplier in a cryptographic accelerator for IPSec authentication. The implementation of MD5 uses full loop unrolling architecture. They used VHDL to implement the cryptographic accelerator on a Xilinx FPGA chip.

Jarvinen et al. [57] proposed an MD5 architecture that can be used for different MD5 implementations. They used their proposed architecture to compare the performance of iterative, full loop unrolling, and partial loop unrolling implementations. The MD5 architecture was implemented on a Xilinx FPGA using VHDL.

3.2.2 SHA-1 Implementations

Most of the work found in the literature focused on FPGA implementation of SHA-1. Zibin and Ning [58] implemented an SHA-1 core that consists of logic blocks used to execute the various operations of SHA-1 (padding, expansion, and compression). These logic blocks are controlled by an FSM (Finite State Machine). They showed the performance of the SHA-1 core on different Altera FPGA chips.

Kitsos et al. [59] proposed an architecture for DSA. This architecture consists of two main parts: an SHA-1 core and an RSA core. The two cores are controlled using a control

unit. They implemented both cores using VHDL and synthesized them on a Xilinx FPGA chip. They showed the performance of each core alone and the performance of the whole system.

Pongyupinpanich et al. [60, 61, 62] implemented SHA-1 on a pipelined architecture to speed up the DSA algorithm. The SHA-1 round computation is divided into four groups, each consisting of 16 compression steps. Each group is implemented on a separate logic block and they run in a pipelined architecture. They concluded that this scheme needs much more gate counts than the architecture without pipelining, but it increases the SHA-1 throughput. They implemented their architecture on a Xilinx FPGA chip using VHDL.

Grembowski et al. [63] implemented SHA-1 for the sake of comparison with SHA-512. Both algorithms were designed using the same methodology and technology. They used VHDL and synthesized the implementations on a Xilinx FPGA chip. They concluded that SHA-512 is faster than SHA-1 but needs more hardware resources.

Sklavos et al. [64] implemented SHA-1 for random number generators (RNGs). The RNG system consists of an Initial Condition Generator (ICG) and an SHA-1 core. The ICG generates the first block to be hashed using the SHA-1 core. The SHA-1 core then hashes the produced block and the resulting message digest is used as a random number. It is also used as a seed for subsequent blocks to be hashed by the SHA-1 core. For this implementation, only one 512-bit block is processed by the SHA-1 core each time. In other words, the maximum message size for this case is 512 bits. The implementation was modeled using VHDL and synthesized on a Xilinx FPGA chip.

3.2.3 RIPEMD-160 Implementations

RIPEMD-160 was proposed and described in [30, 31]. To our knowledge, there is no work that only implemented RIPEMD-160. However, it has been implemented with other hash functions in some work, as will be discussed in the following sections.

3.3 Implementation of Two Hash Functions

A number of literature implemented two hash functions from the three considered. Lin et al. [36] combined a number of encryption and authentication algorithms in an IPSec software module. The hash functions considered in their work are MD5 and SHA-1. They compared the two algorithms in terms of three factors: security against brute-force attacks, security against cryptanalysis, and speed. They concluded that SHA-1 is slower than MD5 for both ESP and AH.

MD5 and RIPEMD-160 were integrated on a unified architecture by Ng et al. [65]. The unified hardware architecture is based on unifying the basic compression steps of the two algorithms in one basic compression step. This unified architecture was modeled in VHDL and synthesized on an Altera FPGA chip.

3.4 Implementation of Three or More Hash Functions

In this section, we review work that implemented three or more hash functions and we indicate which ones of the three considered hash functions were included in each work. Bosselaers et al. [66] implemented the MD4-family hash functions on a Pentium processor in software. They included MD4, MD5, SHA-1, RIPEMD, RIPEMD-128, and RIPEMD-160 in their work.

Roe [67, 68] analyzed the performance of software implementations of some hash functions and encryption algorithms. The hash functions included were MD2, MD4, MD5, SHA-1, and RIPEMD.

An ASIC design of each of MD5, RIPEMD-160, SHA-1, SHA-256, SHA-348, and SHA-512 was given by Satoh and Inoue [69]. Each algorithm was implemented on a different hardware architecture. Two implementations for each algorithm were included, one optimized for area and another optimized for throughput. It should be noted that each hash function was designed alone and that there is no integration among these hash functions.

Dominikus [70] integrated MD5, RIPEMD-160, SHA-1, and SHA-256 in a hash module and implemented it on ASIC and on a Xilinx FPGA chip using VHDL. The hash module is built like a digital processor with a datapath, a control unit, memory, and an input/output unit. These components communicate via some buses. The datapath consists of an expansion block to perform word expansions, a logic block to compute round functions, and an arithmetic unit to calculate the compression steps.

Knag et al. [47] integrated MD5, SHA-1, and HAS-160 in a hash processor. They proposed two architectures. In one architecture, each algorithm is implemented on a different hardware module and the three modules are integrated to form the hash processor. In the other architecture, SHA-1 and HAS-160 are combined on one module and MD5 is implemented on another module. The two architectures were implemented on an Altera FPGA chip.

3.5 Implementation of HMAC

In the above sections, we reviewed work that implemented hash functions without integrating them to HMAC. In this section, we review work that integrated HMAC to some of the considered hash functions.

3.5.1 HMAC with One Hash Function

All work found for HMAC with one hash function focused on FPGA implementation of HMAC-SHA-1. McLoone and McCanny [49] integrated HMAC-SHA-1 and Rijndael on a single-chip cryptographic processor for IPSec. This cryptographic processor consists of two cores, one for encryption using the Rijndael algorithm, and another one for authentication using HMAC-SHA-1. The authentication core consists of an SHA-1 core that implements SHA-1, an HMAC controller to execute the operation required for the HMAC algorithm, which is supported with some additional logic block. The cryptographic processor was implemented on a Xilinx FPGA chip.

Selimis et al. [71] implemented HMAC-SHA-1 to be used for the Wireless Application Protocol (WAP). Their proposed architecture consists of an SHA-1 unit to perform SHA-1, a control unit that coordinates the HMAC operations, some register and logic blocks, and a bus block for communication between the different components of the system. The SHA-1 unit consists of a transformation core that performs the compression steps of SHA-1, a padding unit for padding a message, some registers, and a control unit. The HMAC-SHA-1 unit was implemented on a Xilinx FPGA chip using VHDL.

Michail et al. [72] used a pipelined architecture to implement HMAC-SHA-1. The HMAC pipelined architecture is built using two SHA-1 cores. Each SHA-1 core consists of a 4-stage pipeline, where each stage executes 20 compression steps of SHA-1. The HMAC-SHA-1 architecture was modeled using VHDL and implemented on a Xilinx FPGA chip.

3.5.2 HMAC with Two Hash Function

All work found for HMAC with two hash functions implemented HMAC-MD5 and HMAC-SHA-1. Ha et al. [48] designed an ASIC accelerator for IPSec, which implements HMAC-MD5 and HMAC-SHA-1 along with AES-128/119/256 and TDES. This IPSec accelerator consists of a cryptographic core, a control unit, some register and memory blocks, and some bus and interface blocks. The cryptographic core consists of two parts, an encryption core that implements AES and TDES, and a hash core that implements HMAC-SHA-1 and HMAC-MD5.

Another work considering IPSec was proposed by Lu and Lockwood [50]. They proposed an IPSec implementation on a Xilinx FPGA chip for transport mode. Their implementation includes two cores, an encryption core that implements AES/CBC, and an authentication core that implements either HMAC-SHA-1 or HMAC-MD5. The authentication core consists of an HMAC control logic to control HMAC operations, a hash core that implements one hash function, and some logic blocks to assist the HMAC controller. It should be noted that either HMAC-SHA-1 or HMAC-MD5 is implemented, and hence, there is no integration of the two algorithms on the IPSec implementation.

Wang et al. [73] integrated HMAC-MD5 and HMAC-SHA-1 on an HMAC processor. The HMAC processor consists of three parts, an HMAC controller that manages the data flow between different blocks, a register block, and a hash core. The hash core implements both MD5 and SHA-1. The hash core and the HMAC processor were implemented on ASIC and on an Altera FPGA chip.

3.6 Concluding Remarks

In this chapter, we reviewed previous work that implemented IPSec and hash functions. Table 3.1 summarizes the main features, pros, and cons of the reviewed literature. It only lists previous work that implemented hash functions in hardware, because the focus of this dissertation is on hardware implementations of hash functions.

Table 3.1. *Main features, pros, and cons of the reviewed literature.*

Reference	Hash functions	HMAC	Platform	Design language	Pros	Cons
Deepakumara et al.[56]	MD5	No	FPGA	VHDL	* 2 architectures: iterative and full loop unrolling * Pipelined loading and processing of blocks (using 2 RAMs or buffers)	* Implement only one hash algorithm * Only processing part is implemented
Lee and Kwak [46]	MD5	No	FPGA	VHDL	* Full loop unrolling architecture * Padding and processing components are implemented * Double buffering for pipelined loading of blocks	* Implement only one hash function
Jarvinen et al. [57]	MD5	No	FPGA	VHDL	* Different architectures: iterative, full and partial loop unrolling * Compare block RAM and register implementations * Improve throughput by two approaches: parallel MD5 blocks, and pipelining (for more than one message)	* Implement only one hash algorithm * Only processing and completion parts are implemented

Zibin and Ning [58]	SHA-1	No	FPGA	×	* All SHA-1 steps are implemented * Use 16 registers for the expansion process	* Implement only one hash algorithm * No pipelining
Kitsos et al. [59]	SHA-1	No	FPGA	VHDL	* All SHA-1 steps are implemented	* Implement only one hash algorithm * No pipelining * Padding is done before processing the first block * No details of expansion or throughput
Pongyupinpanich et al. [60, 61, 62]	SHA-1	No	FPGA	VHDL	* All SHA-1 steps are implemented * Pipelined (for different messages) * Use 16 registers for the expansion process (on-the-fly round computation)	* Implement only one hash algorithm * Only one block (maximum message size = 512)
Grembowski et al. [63]	SHA-1, SHA-512	No	FPGA	VHDL	* All SHA-1 steps are implemented * Use 16 registers for the expansion process	* Implement only one hash algorithm at a time (no integration) * No pipelining
Sklavos et al. [64]	SHA-1	No	FPGA	VHDL	* All SHA-1 steps are implemented	* Implement only one hash algorithm * Only one block (maximum message size = 512) * No details of expansion
Ng et al. [65]	MD5, RIPEMD-160	No	FPGA	VHDL	* Unified architecture	* Only one processing line of RIPEMD-160 is implemented * Only processing part is implemented
Satoh and Inoue [69]	MD5, SHA-1, RIPEMD-160, SHA-224/256/348/512	No	ASIC	×	* A dedicated architecture for each hash function * Use 16 registers for the expansion process * Optimized for area or delay	* Each module is alone (no integration) * Only processing part is implemented * No pipelining

Dominikus [70]	MD5, SHA-1, RIPEMD-160, SHA-256	No	FPGA + ASIC	VHDL	* Shared architecture for 4 hash functions (like a digital processor), which allows for scalability * Possible use of parallel arithmetic units	* Only compression steps are reported * No pipelining * Increased number of clock cycles for compression steps
Knag et al. [47]	MD5, SHA-1 HAS-160	No	FPGA	×	* Two architectures: one implements each algorithm alone, and one unifies SHA-1 and HAS-160 (resource sharing) * Use 16 registers for the expansion process	* Only processing parts are implemented
McLoone and McCanny [49]	SHA-1	Yes	FPGA	×	* All SHA-1 steps are implemented * HMAC with fixed key size * Use 16 registers for the expansion process	* Only one hash function (SHA-1 cannot be used alone) * No pipelining
Selimis et al. [71]	SHA-1	Yes	FPGA	×	* All SHA-1 steps are implemented * HMAC with fixed key size	* Only one hash function (SHA-1 cannot be used alone) * No pipelining * No details of the expansion process
Michail et al. [72]	SHA-1	Yes	FPGA	VHDL	* 4-stage pipeline (for 4 different messages) * 2 SHA-1 cores (8 different HMAC messages at a time)	* Only one hash function (SHA-1 cannot be used alone) * 20 registers for each pipeline stage (80 registers) for expansion * No details of the HMAC specifications
Ha et al. [48]	MD5, SHA-1	Yes	ASIC	Verilog	* Integrated architecture for 2 hash functions * HMAC with fixed key size (for IPSec)	* MD5 and SHA-1 cannot be used alone * No pipelining * No details of the expansion process
Lu and Lockwood [50]	MD5, SHA-1	Yes	FPGA	×	* HMAC with fixed key size (for IPSec) * Use 16 registers for SHA-1 expansion process	* Only one hash function (either HMAC-MD5 or HMAC-SHA-1) at a time * Increased number of clock cycles for one compression step * No pipelining

Wang et al. [73]	MD5, SHA-1	Yes	FPGA + ASIC	×	* Integrated architecture For MD5 and SHA-1 * HMAC with general key size * Use 16 registers for SHA-1 expansion process	* Only two hash functions at a time (either HMAC-MD5 and HMAC-SHA-1, or MD5 and SHA-1) * No pipelining

We only include literature considering hardware implementations of hash functions.

"×" means the corresponding information is not reported.

We can see from the table that, to our knowledge, HMAC-RIPEMD-160 was not addressed for FPGA implementation. We can also see from the table that there is no work that integrated HMAC with more than two hash functions. Most of the work reviewed either considered hash functions alone or considered HMAC with hash functions. Work that considered both directions did not implement them on a reconfigurable unit.

Most of the work reviewed showed only architecture for the processing part of the hash functions considered. In addition, most of the work did not apply pipelining or parallelism to enhance the performance of the designed units. Most of the hardware implementations used VHDL to model the design and then synthesized it on ASIC or FPGA.

We chose to save all numerical results of previous work to Chapter 6 in order to compare them with the results of this dissertation.

In the following chapter, the design of a reconfigurable, unified HMAC-hash unit is discussed. We discuss in that chapter the design of a unified hash engine and then the integration of an HMAC unit to it. We discuss the design enhancements applied to improve the performance of the proposed unit.

Chapter 4

Proposed Design

This chapter discusses the design of the proposed HMAC-hash unit in detail. In § 4.1, the design of the unified hash engine is proposed based on a unified algorithm of the three hash functions MD5, SHA-1, and RIPEMD-160. The integration of an HMAC unit to the unified hash engine to form a reconfigurable HMAC-hash unit is discussed in § 4.2. In § 4.3, we discuss the design enhancements applied to improve the performance of our proposed unit. These enhancements include pipelining, parallelism, and key reuse.

4.1 Unified Hash Algorithm

Details of MD5, SHA-1, and RIPEMD-160 hash algorithms have been presented in § 2.3.2, § 2.3.3, and § 2.3.4, respectively. These three hash functions are used alternatively by HMAC and other keyed hash functions. They are based on the same algorithm, MD4, and thus they are quite similar. In § 4.1.1, we present a simplified skeleton of these hash functions that shows the similarities among these algorithms. We highlight the main differences between the three hash functions in § 4.1.2. Studying these similarities and differences, we come up with a unified algorithm that performs the three hash functions. We use this unified algorithm to design a unified hash engine. This is discussed in § 4.1.3.

4.1.1 Hash Functions Similarities

We can summarize the operational similarities among MD5, SHA-1, and RIPEMD-160 in a general skeleton, as shown in Figure 4.1. The three hash algorithms are used to hash a message M of size MS bits, where $0 \leq MS < 2^{64}$. The general skeleton consists of the following four steps [4]:

1. **Initialization:** The constants used by the three hash functions are defined. In addition, certain chaining variables are initialized.
2. **Preprocessing:** The three hash functions under consideration work on 512-bit blocks, one block at a time. Therefore, the message to be hashed has to be of a length divisible by 512 and then segmented into 512-bit blocks. This is done through the following sub-steps:
   (a) **Padding:** The message is appended with a single bit of value '1' followed by l zero bits, where l is the smallest non-negative solution to the equation $MS+1+l$ = 448 mod 512. Then, 64 bits representing the message size MS are appended.
   (b) **Parsing:** After padding, the message is split into N 512-bit blocks M^1 to M^N.
3. **Processing:** This is the core of each algorithm. Each 512-bit block is segmented into 16 32-bit words, X_0 to X_{15}, and processed in one step. This step consists of the following sub-steps:
   (a) **Working variables initialization:** Certain working variables are initialized using the current values of the chaining variables.
   (b) **Compression step:** The working variables are updated in rounds. Each round does the same computation on the working variables in all its steps.
   (c) **Chaining variables update:** The chaining variables are updated.
4. **Completion:** The final hash value is computed by concatenating the chaining variables obtained after processing the last message block M^N.

In § 4.1.3, we show how to implement these steps for the three hash functions in a unified algorithm.

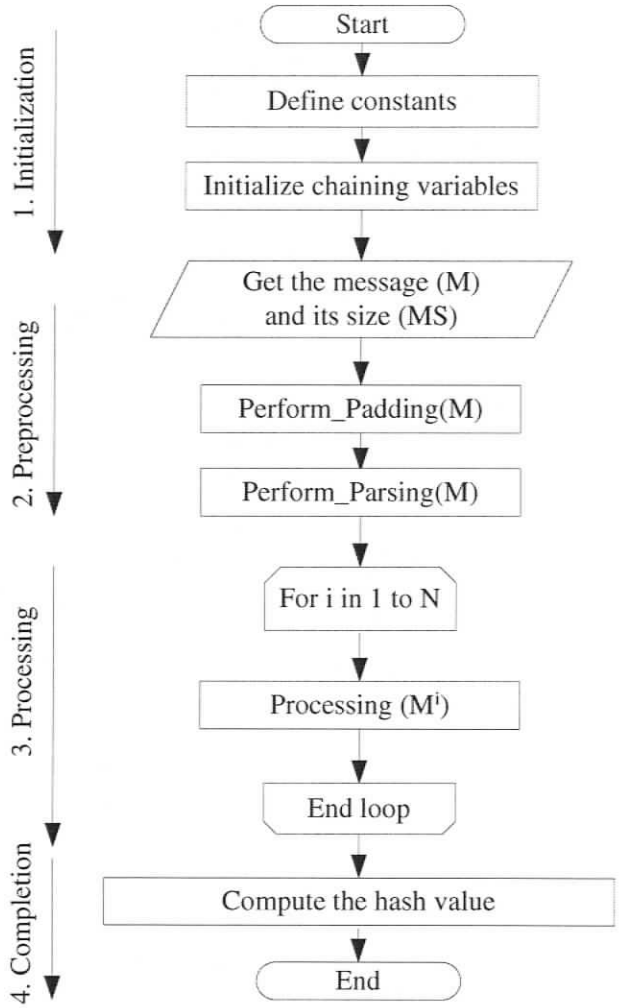

Figure 4.1. *A general skeleton for the three hash algorithms.*

4.1.2 Hash Functions Differences

Although the three hash functions have similarities, there are some differences that must be taken into consideration when designing a unified engine. These differences in the operational steps can be summarized as follows:

1. **Initialization:**
   - MD5 uses 4 chaining variables, $H1$ to $H4$, whereas SHA-1 and RIPEMD-160 use 5 chaining variables, $H1$ to $H5$.
2. **Preprocessing:**
   - The representation of the message words must be in little-endian format for MD5 and RIPEMD-160, whereas they are represented in big-endian format for SHA-1.
3. **Processing:**
   - For each 512-bit message block, MD5 has 4 rounds, each consists of 16 compression steps, SHA-1 has 4 rounds, each consists of 20 compression steps, and RIPEMD-160 has 5 rounds, each consists of 16 compression steps.
   - RIPEMD-160 uses two parallel processing lines (left and right) for each 512-bit message block, whereas there is only one processing line used in each of MD5 and SHA-1.
   - The 16 32-bit message words, X_0 to X_{15} are expanded to 80 words, X_0 to X_{79} for SHA-1. Then, in each compression step, one word is referenced in a sequential order. However, MD5 and RIPEMD-160 use only the 16 words, X_0 to X_{15}. In each compression step j, one word is referenced according to the access order constants ($z_m[j]$ for MD5 and $z_{LR}[j]$ and $z_{RR}[j]$ for RIPEMD-160).
   - MD5 uses 4 working variables, A to D, SHA-1 uses 5 working variables, A to E, and RIPEMD-160 uses 10 working variables, AL to EL for the left processing line and AR to ER for the right processing line.

- Each hash function has its own way to update the working variables in compression steps, as will be discussed in the following section.
- After processing each 512-bit message block, MD5 and SHA-1 update the chaining variables in the same way, whereas RIPEMD-160 uses a different way to update them, as will be discussed in the following section.

4. **Completion:**
   - The representation of the final hash value must be in little-endian format for MD5 and RIPEMD-160, whereas it is represented in big-endian format for SHA-1.
   - The final hash value of MD5 is 128-bit long, whereas it is 160-bit long for each of SHA-1 and RIPEMD-160.

4.1.3 The Unified Algorithm

Based on the similarities and differences between the three hash functions, we propose a unified algorithm and then implement this algorithm on a unified, reconfigurable architecture.

The unified algorithm is shown in Figure 4.2. Similar to the general skeleton described in § 4.1.1, the unified algorithm consists of four main steps. These four steps are described as follows:

1. **Initialization:** In this step, we define all constants required for the three hash algorithms (as described in § 2.3). These constants include initial chaining values (IVs), order of accessing message words, additive constants, and the number of bits for rotate-left operation in each step.

   Careful inspection of the three hash algorithms shows that the first four IVs are the same for all of them, and the fifth IV is the same for SHA-1 and RIPEMD-160. Therefore, the IVs are used to initialize five chaining variables, $H1$ to $H5$, as follows:

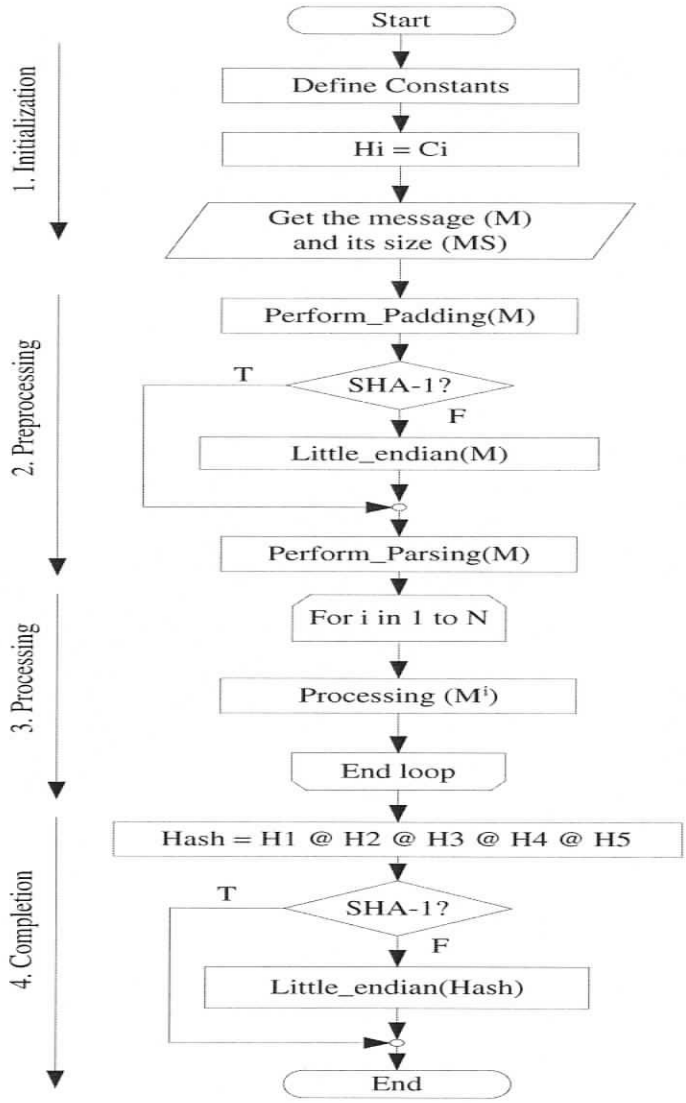

Figure 4.2. *The proposed unified hash algorithm.*

$$H1 = C1 = \text{0x67452301}$$
$$H2 = C2 = \text{0xefcdab89}$$
$$H3 = C3 = \text{0x98badcfe}$$
$$H4 = C4 = \text{0x10325476}$$
$$H5 = C5 = \text{0xc3d2e1f0}$$

For the other constants, we use the terms defined in § 2.3.

2. **Preprocessing:** This step consists of message padding and parsing. The general padding idea was described in the general skeleton described in § 4.1.1. However, there are three cases of the padding process, depending on the message size, MS. Figure 4.3 illustrates these padding cases. Let $count$ be the remainder of dividing the message size MS by 512. Figure 4.3 (a) shows the padding case if $count = 0$. The padding process is done on a new message block. In other words, the original message will be the same and a new 512-bit block is added to the message. Figure 4.3 (b) shows the padding case if $0 < count \leq 447$. The padding process is done on the original message to make its length divisible by 512. Figure 4.3 (c) shows the padding case if $447 < count < 512$. In this case, the remaining bits for padding is less than or equal to 64 bits, which is not enough for the whole padding process. Therefore, the padding process is done on the original message to make its length divisible by 512. Then, an additional block is added for the rest of the padding process.

   The representation of the message depends on the algorithm, as was shown in Figure 4.2. If the algorithm is SHA-1, the message is represented in big-endian format. Otherwise, it is represented in little-endian format. The little-endian conversion is done on each word (32 bits) of the message as follows:

   Let W be a 32-bit word consisting of four bytes $B1$ to $B4$. If the big-endian representation of W is $B1@B2@B3@B4$, then

$$\text{Little\_endian}(W) = B4@B3@B2@B1.$$

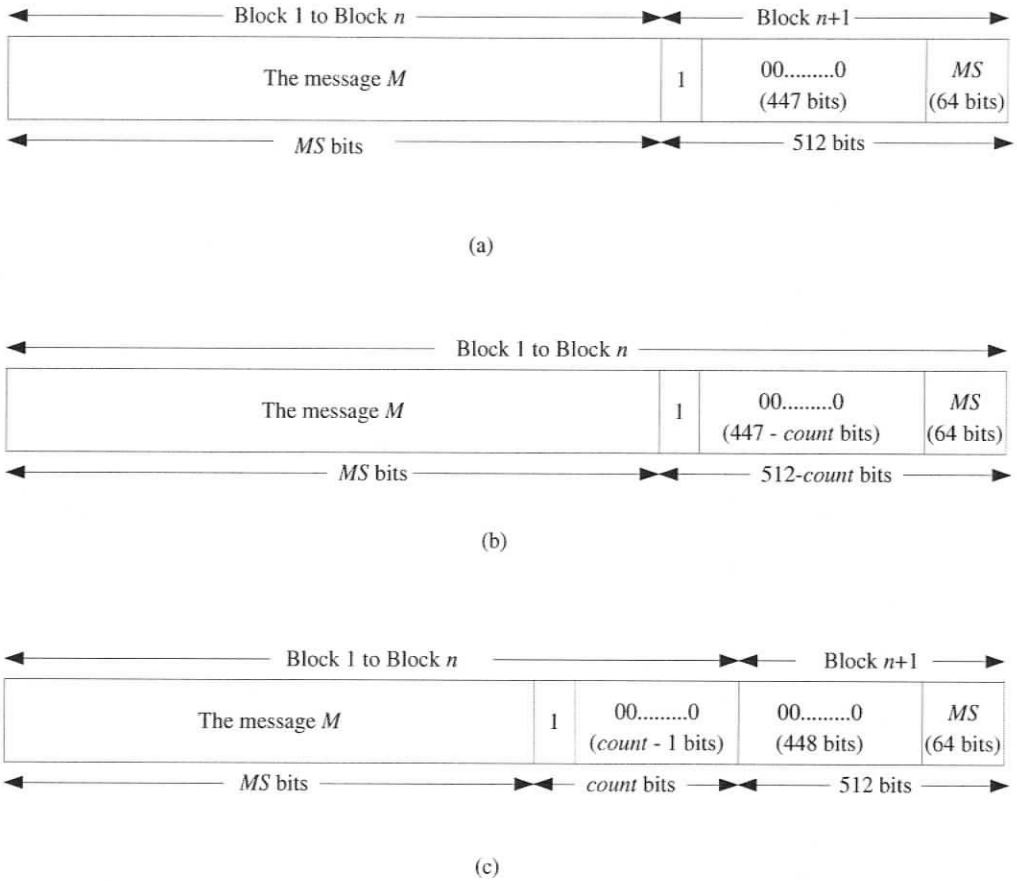

Figure 4.3. *Different cases of the padding step. (a) Padding when* $count = 0$. *(b) Padding when* $0 < count \leq 447$. *(c) Padding when* $447 < count < 512$.

MS = *the original message size,* $count = MS \bmod 512$, *and* n = *the integer value of* $MS \div 512$.

For example, let W = 0x01234567 in big-endian representation. Then,

$$\text{Little\_endian}(W) = \text{0x67452301}.$$

After padding, the message is segmented into N 512-bit blocks, where $N = n + 1$ for cases (a) and (c), and $N = n$ for case (b) of Figure 4.3.

3. **Processing:** Most of the differences between the three hash functions exist in this step. Figure 4.4 illustrates the processing section of the unified algorithm. The first step is to segment the message block under processing into 16 32-bit words, X_0 to X_{15}. If the hash function selected is SHA-1, an expansion process is performed. This process expands the 16 words, X_0 to X_{15}, into 80 words, X_0 to X_{79}. Then, five working variables, A to E, are initialized with the values of the chaining variables $H1$ to $H5$, respectively. Following this, 80 compression steps are performed. However, if the algorithm is MD5, the processing stops after 64 compression steps.

   In each compression step, a temporary variable T is computed and used to update the working variables. T is computed using the function *Compute_T*, which involves an addition modulo 2^{32} of five arguments, $T1$ to $T5$. The first four arguments, $T1$ to $T4$, are defined for all compression steps as follows:

$$T1 = \begin{cases} ROTL^5(A), & \text{for SHA-1} \\ A, & \text{for MD5 and RIPEMD-160} \end{cases}$$

$$T2 = \begin{cases} E, & \text{for SHA-1} \\ 0, & \text{for MD5 and RIPEMD-160} \end{cases}$$

$$T3 = \begin{cases} X_{z_m[j]}, & \text{for MD5} \\ X_j, & \text{for SHA-1} \\ X_{z_{RL}[j]}, & \text{for RIPEMD-160} \end{cases}$$

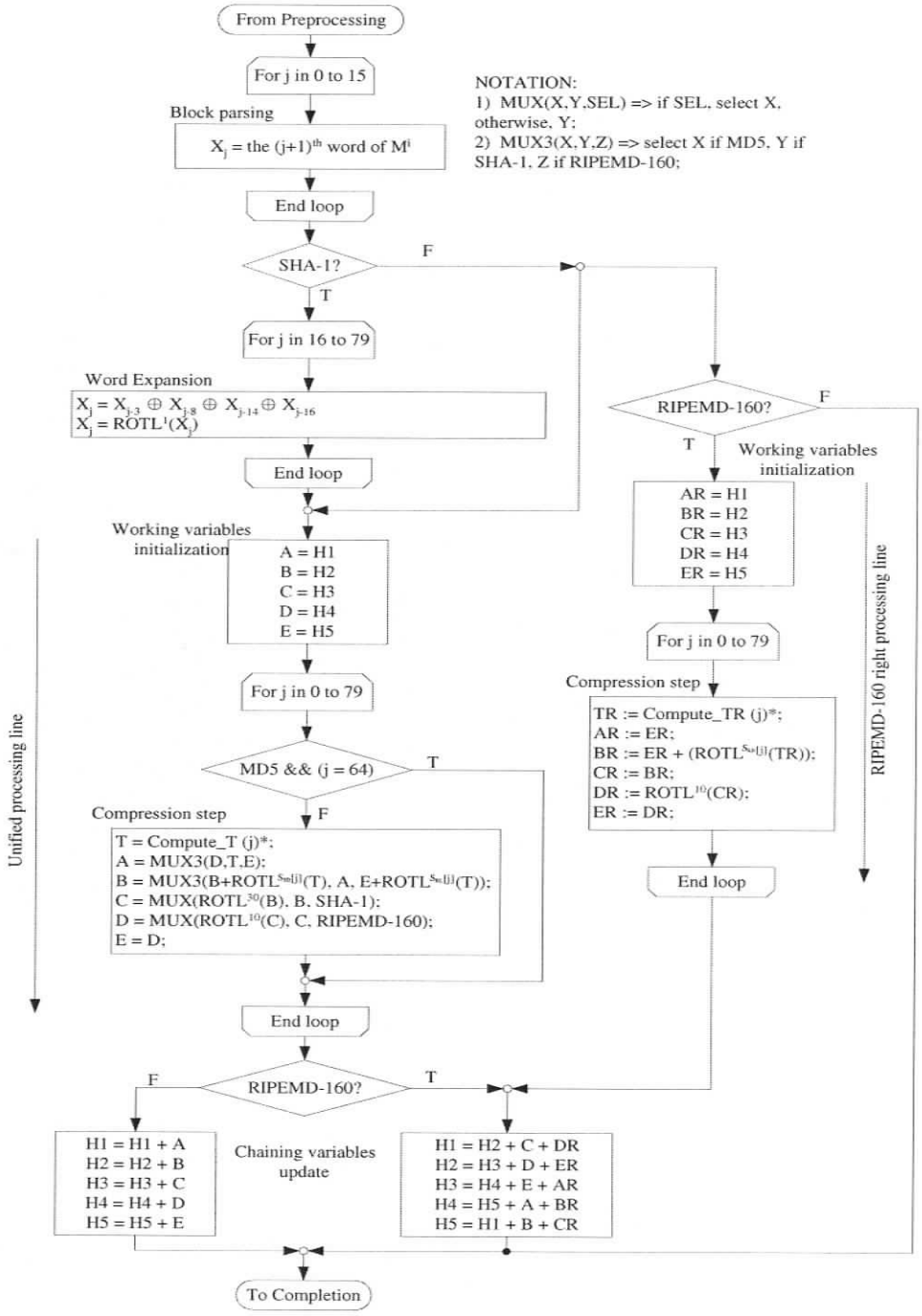

Figure 4.4. *Processing section of the unified hash algorithm.*

** Compute_T and Compute_TR are explained in the text.*

$$T4 = \begin{cases} y_m[j], & \text{for MD5} \\ y_s[j \ mod \ 20], & \text{for SHA-1} \\ y_{RL}[j \ mod \ 16], & \text{for RIPEMD-160} \end{cases}$$

where j is the compression step number with $0 \leq j \leq 79$.

The fifth argument, $T5$, is a round function, f, whose inputs are the working variables B, C, and D. In the original hash algorithms, every round in each algorithm has a different function. In our unified algorithm, we compressed these functions into four functions, $f1, f2, f3$, and $f4$, such that:

$f1(u, v, w) = (u \ \& \ v) \mid (\overline{u} \ \& \ w)$,

$f2(u, v, w) = (u \ \& \ v) \mid (u \ \& \ w) \mid (v \ \& \ w)$,

$f3(u, v, w) = u \oplus v \oplus w$, and

$f4(u, v, w) = u \oplus (v \mid \overline{w})$,

where u, v, and w are the three inputs to the round function in that order. These four functions are used alternatively in all rounds of the unified algorithm with different orders of B, C, and D. As we mentioned in § 4.1.2, the three hash functions differ in the number of rounds and the number of compression steps per round. In order to combine the three algorithms, the number of steps in each round has to be modified. This modification resulted in seven rounds for MD5, and eight rounds for each of SHA-1 and RIPEMD-160. Table 4.1 shows the rounds we used, their limits, and the round function used in each round for each of the three hash functions.

For RIPEMD-160, a parallel processing line is required, which is called right processing line. This line is only activated if the required algorithm is RIPEMD-160. The deactivation of this block may save power consumption when one of the other two hash functions is selected. This line has the same processing steps as the left RIPEMD-160 processing line but using different working variables (AR to ER), different constants (s_{RR}, z_{RR}, and y_{RR}) , and a different temporary register, TR, which

Table 4.1. *Round functions of the unified algorithm.*

Round	Algorithm	Round function
$0 \leq j \leq 15$	MD5 SHA-1	$f1(B, C, D)$
	RIPEMD-160	$f3(B, C, D)$
$16 \leq j \leq 19$	MD5	$f1(D, B, C)$
	SHA-1 RIPEMD-160	$f1(B, C, D)$
$20 \leq j \leq 31$	MD5	$f1(D, B, C)$
	SHA-1	$f3(B, C, D)$
	RIPEMD-160	$f1(B, C, D)$
$32 \leq j \leq 39$	MD5 SHA-1	$f3(B, C, D)$
	RIPEMD-160	$f4(D, B, C)$
$40 \leq j \leq 47$	MD5	$f3(B, C, D)$
	SHA-1	$f2(B, C, D)$
	RIPEMD-160	$f4(D, B, C)$
$48 \leq j \leq 59$	MD5	$f4(C, B, D)$
	SHA-1	$f2(B, C, D)$
	RIPEMD-160	$f1(D, B, C)$
$60 \leq j \leq 63$	MD5	$f4(C, B, D)$
	SHA-1	$f3(B, C, D)$
	RIPEMD-160	$f1(D, B, C)$
$64 \leq j \leq 79$	SHA-1	$f3(B, C, D)$
	RIPEMD-160	$f4(B, C, D)$

is computed using the function *Compute_TR*. The round functions for RIPEMD-160 right processing line are used in the reverse order than the left processing line, as shown in Table 4.2.

Table 4.2. *Round functions of the two processing lines of RIPEMD-160.*

Round	Left line	Right line
$0 \leq j \leq 15$	$f3(B, C, D)$	$f4(BR, CR, DR)$
$16 \leq j \leq 31$	$f1(B, C, D)$	$f1(DR, BR, CR)$
$32 \leq j \leq 47$	$f4(D, B, C)$	$f4(DR, BR, CR)$
$48 \leq j \leq 63$	$f1(D, B, C)$	$f1(BR, CR, DR)$
$64 \leq j \leq 79$	$f4(B, C, D)$	$f3(BR, CR, DR)$

The final step in each processing iteration is to update the chaining variables. For MD5 and SHA-1, the update is simply done by adding the values of the working variables A to E to the current values of $H1$ to $H5$, respectively. For RIPEMD-160, the update is more complicated and is computed as shown in Figure 4.4.

4. **Completion:** The final step of the unified algorithm is to compute the hash value by concatenating the five chaining variables $H1$ to $H5$ and represent it in the appropriate endian format. This algorithm produces a 160-bit hash, and if the required hash function is MD5, the most significant 128 bits are selected.

It should be noted that in steps 1, 2, and 4 of the unified algorithm, the only difference between the three hash functions is in the little-endian representation of the message and hash value for MD5 and RIPEMD-160.

The above described unified hash algorithm is used to design a hash engine. In the following section, we discuss how this hash engine is integrated with an HMAC unit to form the proposed HMAC-hash unit.

4.2 Integrated HMAC-Hash Unit

In this section, we discuss the implementation of the proposed HMAC-hash unit. In the following subsection, the architecture of the HMAC-hash unit is discussed. In § 4.2.2, we discuss how we designed the proposed unit for reconfigurability. In § 4.2.3, the implementation of the HMAC algorithm for fixed key size is discussed.

4.2.1 HMAC-Hash Unit Architecture

Figure 4.5 shows a block diagram of the proposed HMAC-hash unit, and Table 4.3 summarizes the functionality of the input/output signals shown in the figure. The maximum input size of the designed HMAC-hash unit is 32 bits. So, 2 clock cycles are required to get the message and key sizes and 16 clock cycles are required to receive each 512-bit block.

We designed the HMAC-hash unit as two sub-units, an HMAC sub-unit and a hash sub-unit. The HMAC sub-unit receives the message size (MS), key size (KS), and then the message and key in 512-bit blocks, one block at a time. It passes each block to the hash sub-unit, which is used for hashing the message (or the key if required) using the unified hash algorithm discussed in § 4.1. The $Mode$ and Alg signals are used to reconfigure the HMAC sub-unit and select one hash function, as will be discussed in § 4.2.2.

The port MAC is used to output the result. The output size of the HMAC sub-unit is 160 bits. If the output is to be truncated (as described in the HMAC algorithm, Algorithm 2.1, in § 2.3.1), the most significant bits of the output are selected as required. Since this step (the truncation of the output) does not affect the performance of the HMAC sub-unit, we assume that the communicating entity with the HMAC-hash unit has the capability of selecting the required number of output bits.

We avoided receiving the whole message and buffering it because the maximum message size is 2^{64}, which needs a large memory for buffering. This arrangement applies also in the case of receiving a key with size greater than 512 bits. Therefore, we assume that there is a handshaking mechanism between the HMAC-hash unit and the entity communicating

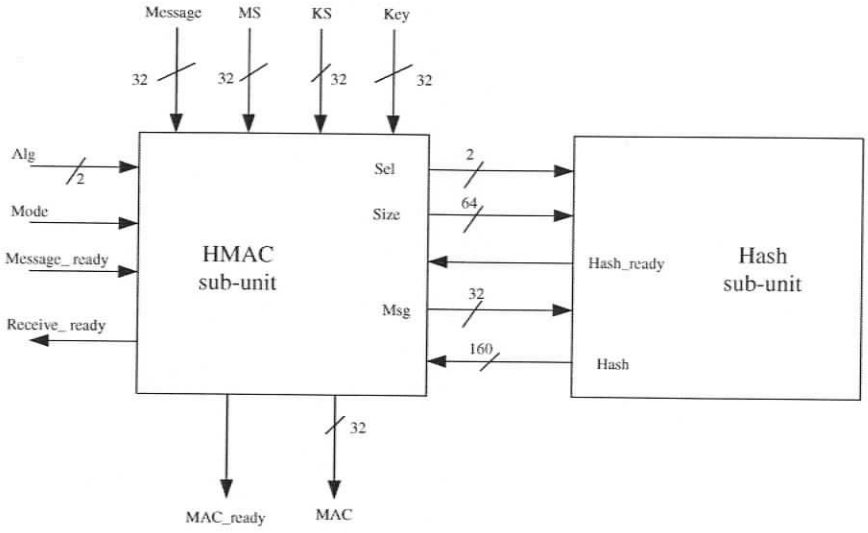

Figure 4.5. *Block diagram of the HMAC-hash unit.*

Table 4.3. *Input/output signals of the HMAC-hash unit and their functionality.*

Signal	Direction	Function
Mode	In	Activates HMAC (1) or deactivates it (0)
Alg[1:0]	In	Selects the hash function to be used (see Table 4.4)
Message_ready	In	Indicates that a new message is available
KS[31:0]	In	A 32-bit block of the key size (in bits)
MS[31:0]	In	A 32-bit block of the message size (in bits)
Key[31:0]	In	A 32-bit block of the key
Message[31:0]	In	A 32-bit block of the message
Receive_ready	Out	Indicates that HMAC sub-unit is ready to receive a message block
MAC_ ready	Out	Indicates that the MAC is ready for output
MAC[31:0]	Out	A 32-bit block of the MAC

with it, as shown in Figure 4.6.

On power-on, the hash sub-unit sets the $Hash\_ready$ signal to '1', which means that the hash sub-unit is ready to receive a block of the message. Then, The HMAC sub-unit waits until the communicating entity sets the $Message\_ready$ signal to '1'. Then, the HMAC sub-unit sets the $Receive\_ready$ signal to '1', starts receiving a message block, and passes this message block to the hash sub-unit. Then, it sets the $Receive\_ready$ signal to '0' to allow enough time for the hash sub-unit to process the received block. When the hash sub-unit is ready again, another block of the message, if any, is received in the same way. After processing all message blocks and when the MAC is ready, the HMAC sub-unit sets the MAC_ready signal to '1' and outputs the MAC.

4.2.2 Reconfigurability

The HMAC sub-unit is reconfigured by 2 inputs: $Mode$ (1 bit) and Alg (2 bits) as shown in Table 4.4. The $Mode$ bit decides whether the HMAC algorithm is to be included or only a hash function is executed. In both cases, the same output port MAC is used to output the result.

Table 4.4. *Reconfigurability of the HMAC sub-unit.*

Mode	Alg	Selected hash function
0	00	HMAC-MD5 (RFC 2403 [5])
0	01	HMAC-SHA-1 (RFC 2404 [6])
0	10	HMAC-RIPEMD-160 (RFC 2857 [7])
0	11	Not used
1	00	MD5 (RFC 1321 [27])
1	01	SHA-1 (RFC 3174 [29])
1	10	RIPEMD-160 [30]
1	11	Not used

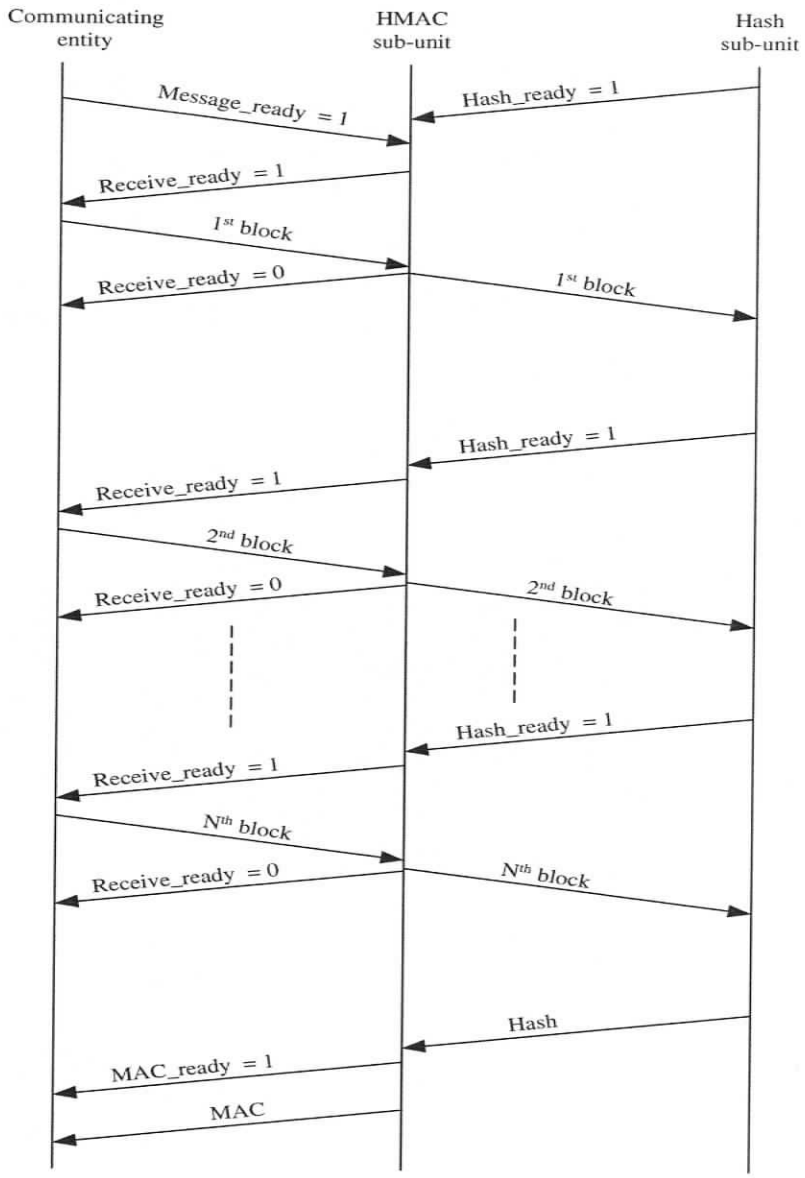

Figure 4.6. *Handshaking between the HMAC-hash unit and the communicating entity.*

As was shown in Figure 4.5, the hash sub-unit gets a two-bit input, Sel, from the HMAC sub-unit, which is a copy of the signal Alg. This signal selects the hash function to be used. The hash sub-unit gets also the message size, $Size$, and the message, Msg, one 512-bit block at a time. If the $Mode$ bit is zero and the key needs to be hashed, the key is sent to the hash sub-unit as a regular message.

While the reconfigurability applied here is static, where one configuration is used for the six algorithms, it can be considered as a functional reconfigurability, since the functionality of the hardware is modified at runtime using the control signals $Mode$ and Alg. In this case, runtime reconfigurability is applied while eliminating the overhead of reloading a new configuration each time.

4.2.3 HMAC Implementation for Fixed Key Size

The HMAC-hash unit described in § 4.2.1 implements the general HMAC algorithm as described in Algorithm 2.1 (§ 2.3.1) [25, 26], which allows keys of any length. For example, HMAC-MD5 specified in RFC 2085 [74] recommends the support of long key lengths. However, in [5, 6, 7], the key size is restricted to 128 bits for HMAC-MD5 and 160 bits for each of HMAC-SHA-1 and HMAC-RIPEMD-160. Therefore, we implemented another version of the HMAC-hash unit using a key size of 160 bits. If the hash function selected is HMAC-MD5, the least significant 32 bits of the key are filled with zeros. Since the key size is fixed, there is no need to get the key size as an input to the HMAC sub-unit.

This implementation eliminates the key resizing step from the HMAC algorithm (Equation 2.2). Instead, K_0 is created by filling the least significant 352 bits (512 - 160) with zeros. We can get benefit of this feature by noting the following (referring to Figure 4.5 and to the HMAC algorithm (Algorithm 2.1)):

- The hash sub-unit receives the message one 32-bit word at a time, starting from the most significant word.
- $ipad$ and $opad$ can be represented as repeated 32-bit words.

- $V2$ and $V5$ are formed by XORing K_0 with $ipad$ and $opad$, respectively, and then concatenated to the message from the most significant position.
- Hashing a message is done in a pipelined fashion, as will be discussed in § 4.3.1.

Using the above implementation facts, the HMAC algorithm using fixed key size can be implemented as follows:
Let K be the 160-bit key received such that:

$$K = K1@K2@K3@K4@K5,$$

where $K1$ to $K5$ are 32-bit words. Let $ipad_{32}$ and $opad_{32}$ be a 32-bit word of $ipad$ and $opad$, respectively.

Figure 4.7 shows how words are sent to the hash sub-unit. Figure 4.7 (a) shows the first block of $V2$ in Algorithm 2.1 (which is $V1$). This block consists of 16 32-bit words. These words are sent to the hash sub-unit in order, one word at a time, starting from the left most word. After this block is sent, the message M is sent to the hash sub-unit, one block at a time. This implements steps 2, 3, and 4 of the HMAC algorithm (Algorithm 2.1).

$V5$ is sent to the hash sub-unit in a similar way, as shown in Figure 4.7 (b). $V4$, is sent first to the hash sub-unit, followed by $V3$. This implements steps 5, 6, and 7 of Algorithm 2.1.

4.3 Design Enhancements

We applied speedup techniques, such as pipelining and parallelism, to improve the performance of the proposed unit. In addition, we propose a key reuse technique to enhance the throughput of the proposed unit when a key is reused for successive messages.

4.3.1 Pipelining

In order to increase the throughput of the HMAC-hash unit, a simple pipeline arrangement is used, as shown in Figure 4.8. This pipeline consists of two stages. The first stage is used

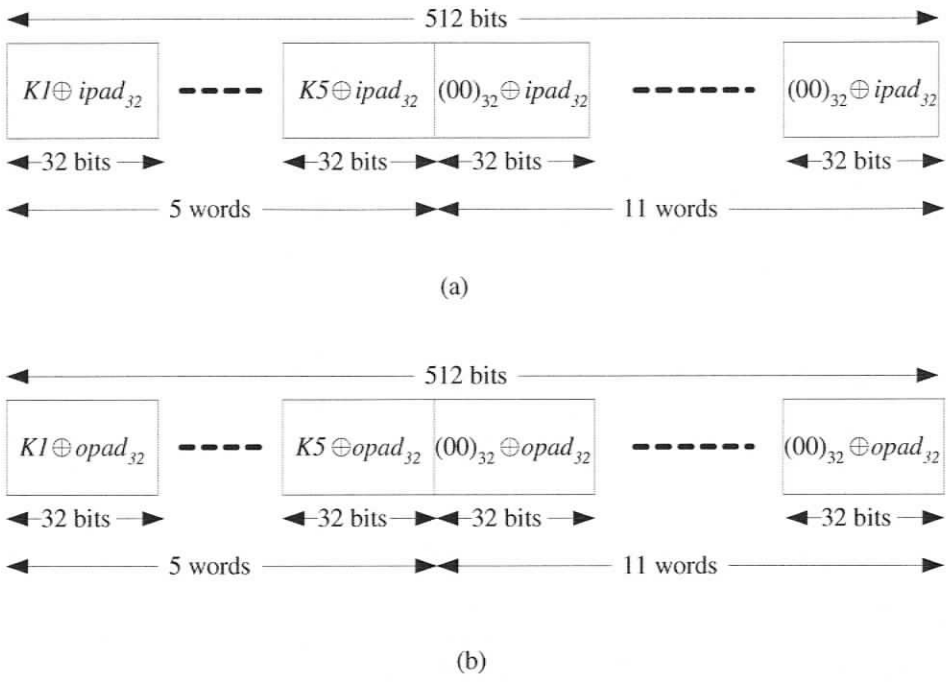

Figure 4.7. *Formatting the first block before sending it to the hash sub-unit in case of using a fixed key size. (a) First block of $V2$ (step 2 of Algorithm 2.1). (b) First block of $V5$ (step 6 of Algorithm 2.1)*

for preprocessing, which includes receiving a message block and segmenting it into 16 32-bit registers (X_0 to X_{15}), preparing it in the appropriate endian format for the required hash function, and performing the padding process on the last message block. This stage takes t_2 clock cycles, which ranges from 18 to 21 depending on the hash function and the padding case, which depends on the message size (see § 4.1.3). The initialization step is executed in parallel with the preprocessing of the first block, as will be discussed in § 4.3.2. The initialization stage takes t_1 clock cycles, which equals to one.

The second pipeline stage is responsible for the processing of the message block using the 16 32-bit registers. This stage takes t_3 clock cycles, which is 130 clock cycles for MD5 and 162 clock cycles for each of SHA-1 and RIPEMD-160. During processing a block, the

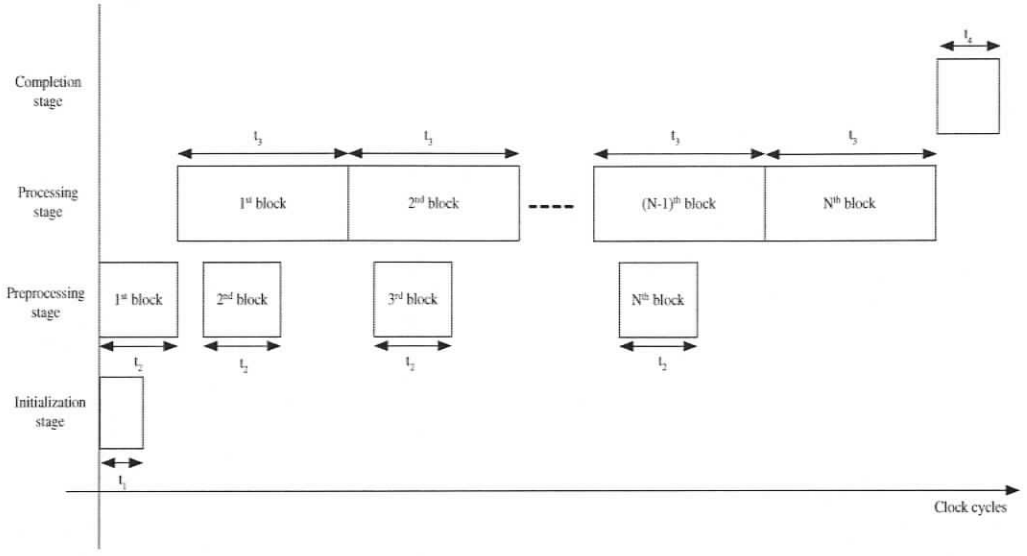

Figure 4.8. *Task scheduling for pipelining the preprocessing and processing stages. The preprocessing stage waits for some registers to be free before starting preprocessing the next block.*

preprocessing stage starts preparing the next block. In order to do this, we used another set of 16 32-bit registers, R_0 to R_{15}. The values of the these registers are copied from X_0 to X_{15}, respectively. This is done in parallel with the processing of the message block. After registers X_0 to X_{15} are copied and become free, the preprocessing stage starts preparing the next block.

Since the preprocessing time is much shorter than the processing time, the preprocessing stage stalls waiting for the processing stage to finish processing the current block and start processing the next prepared block; then it starts preprocessing the next block, if any.

After processing all message blocks, a completion stage is required to prepare the hash value for output. This stage takes t_4 clock cycles, which is 2 clock cycles for SHA-1 and 5 clock cycles for each of MD5 and RIPEMD-160.

4.3.2 Parallelism

Although the algorithm described in § 4.1.3 is shown sequentially in Figures 4.2 and 4.4, we could utilize some inherent parallelism to reduce the time required. Referring to Figure 4.9, this utilization can be summarized as follows:

1. The initialization step, which consists of defining constants and initializing the chaining variables $H1$ to $H5$, can be done in parallel with the preprocessing (message padding and parsing). This is shown in Figure 4.9 (a).
2. The right processing line of RIPEMD-160 can run in parallel with the left processing line (which is the unified processing line). This is shown in Figure 4.9 (b).
3. Since SHA-1 accesses the message words sequentially, we could use only the original 16 words, and then perform the expansion process in parallel with the processing. This is shown in Figure 4.9 (c). As we mentioned in § 4.3.1, the 16 32-bit words X_0 to X_{15} are copied to R_0 to R_{15}, respectively, for pipelining purpose. Then, each expanded word R_i will be stored in $R_{i \ mod \ 16}$ after SHA-1 has already accessed $R_{i \ mod \ 16}$. Table 4.5 shows the 16 registers used to store the 80 words and the words stored in each register.

Figure 4.10 shows the unified algorithm after applying parallelism. It combines Figures 4.2, and 4.4, focusing on the parts implemented in parallel, as was shown in Figure 4.9.

4.3.3 Key Reuse

In order to further improve the designed HMAC-hash unit, we propose a key reuse mechanism. This mechanism is useful when the same key is used for successive messages. In that case, there is no need to receive the key and resize it. Instead, the HMAC algorithm uses the available resized key that has been used for the last message. For this purpose, we add a new one-bit input signal called $Same\_key$. When $Same\_key$ = '1', receiving and resizing the key are deactivated, and the old resized key is used. When $Same\_key$ = '0', a new key has to be received and resized.

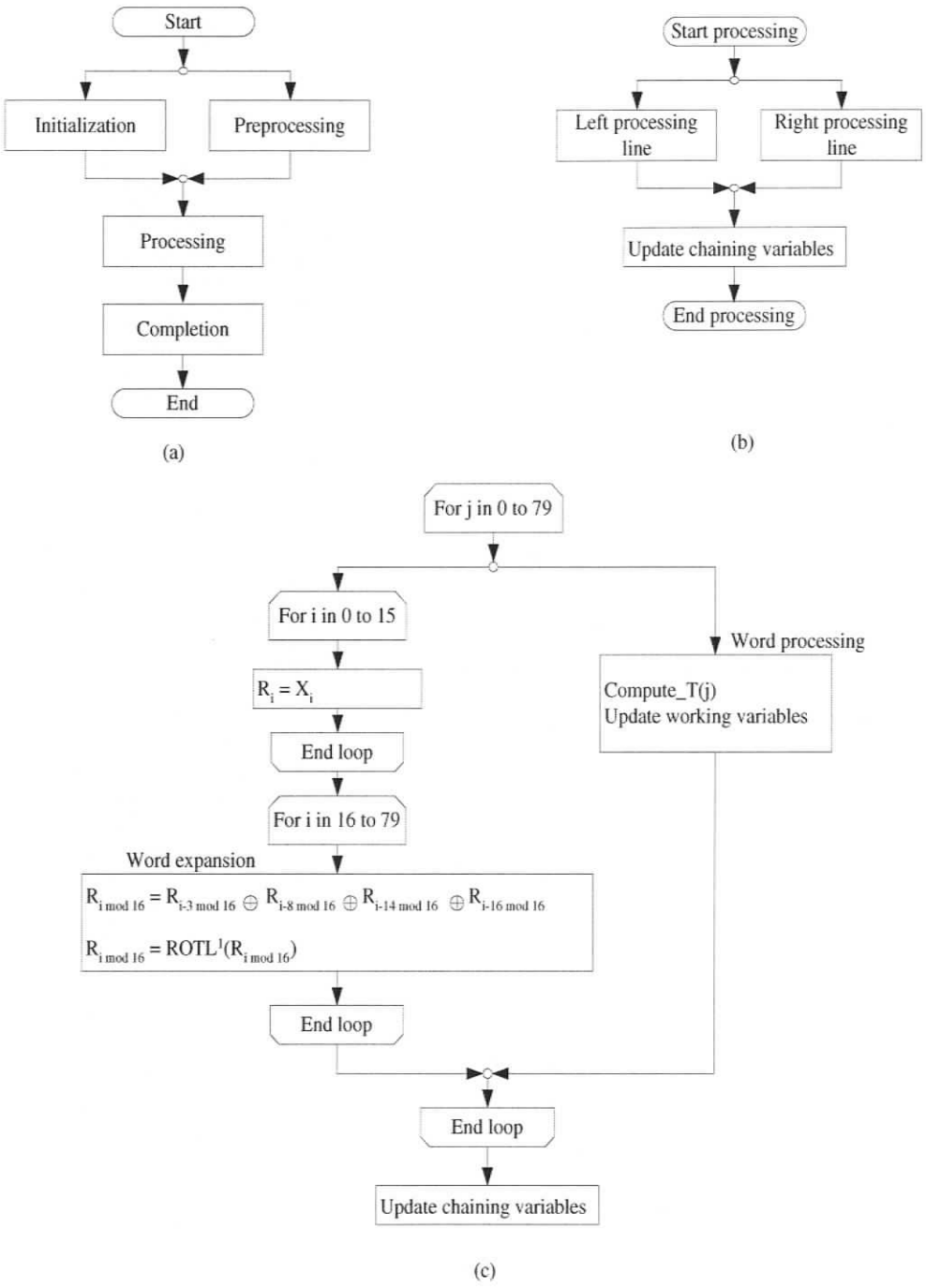

Figure 4.9. *Utilizing parallelism in implementing the unified algorithm. (a) Running the initialization and preprocessing in parallel. (b) Running the two RIPEMD-160 processing lines in parallel. (c) Running SHA-1 word expansion and word processing in parallel.*

Table 4.5. *Storing the 80 expanded words of SHA-1 in 16 registers.*

Register number	Word number
0	0, 16, 32, 48, 64
1	1, 17, 33, 49, 65
2	2, 18, 34, 50, 66
3	3, 19, 35, 51, 67
4	4, 20, 36, 52, 68
5	5, 21, 37, 53, 69
6	6, 22, 38, 54, 70
7	7, 23, 39, 55, 71
8	8, 24, 40, 56, 72
9	9, 25, 41, 57, 73
10	10, 26, 42, 58, 74
11	11, 27, 43, 59, 75
12	12, 28, 44, 60, 76
13	13, 29, 45, 61, 77
14	14, 30, 46, 62, 78
15	15, 31, 47, 63, 79

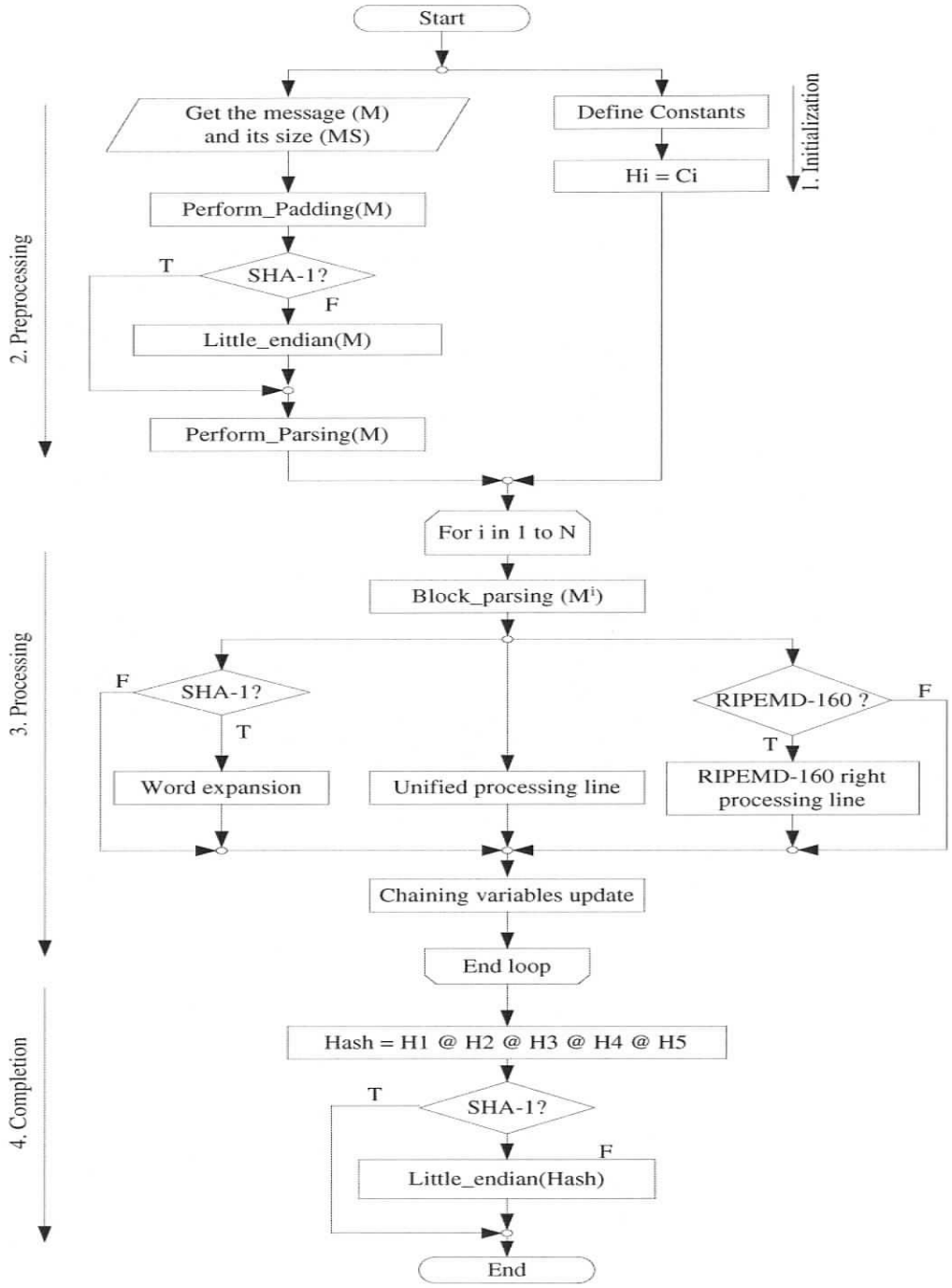

Figure 4.10. *The unified algorithm with parallelism.*

However, we should be careful when using this mechanism. If the key size is greater than 512, the key needs to be hashed for resizing. If the hash function selected for the current message is not the same as that used with the previous message, the resized key for the current message will be different than the resized key used for the previous message, even if the key is the same before resizing.

Table 4.6 illustrates the different cases for the key reuse mechanism. If $Same\_key$ = '0', the key has to be resized anyway. If $Same\_key$ = '1', we have to check the hash function to be used. If it is the same, the old key is used. If it is different, the old key is used only if the key size is less than or equal to 512. Otherwise, a new key has to be received and resized.

Table 4.6. *Different cases for key resizing when using key reuse.*

Same_key	Hash function selected	Key size	Key resizing
0	any	any	Yes
1	same as previous message	any	No
1	different than previous message	≤ 512	No
1	different than previous message	> 512	Yes

If we combine the key reuse mechanism with the possible HMAC implementations described in § 4.2.3, we have the following possible implementations:

(a) General key size with no key reuse. We call this implementation $D1$.

(b) General key size with key reuse. We call this implementation $D2$.

(c) Fixed key size with no key reuse. We call this implementation $D3$.

(d) Fixed key size with key reuse. We call this implementation $D4$.

Figure 4.11 shows the input/output signals of the above four implementations of the HMAC-hash unit. Implementations with no key reuse (Figure 4.11 (a) and (c)) do not include the $Same\_key$ signal. Implementations with fixed key size (Figure 4.11 (c) and

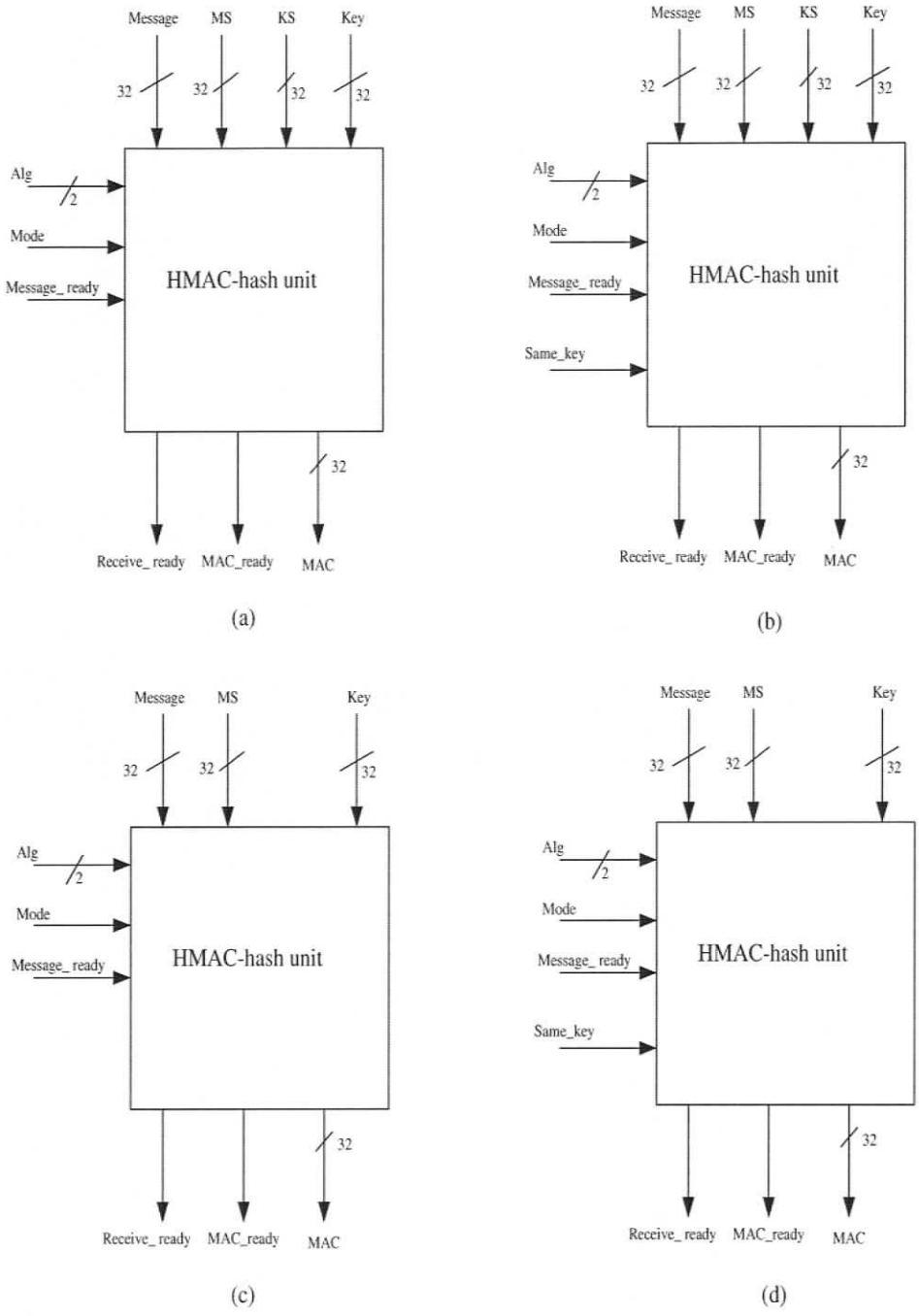

Figure 4.11. *Four implementations of the HMAC-hash unit and their input/output. (a) General key size with no key reuse, D1. (b) General key size with key reuse, D2. (c) Fixed key size with no key reuse, D3. (d) Fixed key size with key reuse, D4.*

(d)) do not need the KS signal. In Chapter 6, we will compare the performance of these four implementations.

4.4 Concluding Remarks

In this chapter, we discussed the design of the proposed HMAC-hash unit. We designed a unified hash engine that implements any one of MD5, SHA-1, and RIPEMD-160. This work has been published in [75]. We integrated an HMAC sub-unit to the unified hash engine to form a reconfigurable, unified HMAC-hash unit that implements any one of MD5, SHA-1, RIPEMD-160, HMAC-MD5, HMAC-SHA-1, and HMAC-RIPEMD-160. This work has been accepted for publication in [76]. We applied pipelining and parallelism to improve the design of the proposed unit. We also proposed a key reuse technique to further improve the design.

In the following chapter, the experimental setup is discussed. We describe the tools we used to design the proposed unit discussed in this chapter. Then, we discuss the test strategy used to test the designed unit. We then discuss the design space exploration applied to the design of the proposed unit, and give examples of using Handel-C to design it.

Chapter 5

Experimental Setup

In this chapter, the experimental setup to model, implement, and test the proposed design is discussed. In § 5.1, the experimental tools used to model, implement, and analyze the performance of the proposed unit are briefly described. The design flow using these tools and how it was applied to the modeling of the proposed unit is discussed in § 5.2. In § 5.3, the test strategy we followed to test the proposed unit is described. We conducted a design space exploration in order to get the most optimal results for the performance of the proposed unit. This is discussed in § 5.4. In § 5.5, examples of using Handel-C to design the proposed HMAC-hash unit are given.

5.1 Experimental Tools

The design methodology we followed to design the proposed HMAC-hash unit is assisted with some experimental tools. The highest level of abstraction uses Handel-C langauge, which is discussed in § 5.1.1. Handel-C codes are compiled and translated to EDIF (Electrical Design Interchange Format) using Celoxica Design Kit (DK), which is discussed in § 5.1.2. EDIF data is mapped to FPGA chips using Xilinx ISE tools, which are discussed in § 5.1.3.

5.1.1 Handel-C

Handel-C[1] [18, 19] is a high level language based on ANSI-C. It is designed to enable direct compilation of a design from high level descriptions to FPGA logic or RTL descriptions. It allows designers to import algorithms written in C and use them to create a rapid implementation and optimization path to hardware.

The advantage of Handel-C over conventional C is that it is fully synthesizable and that it supports parallelism. Most of the constructs available in ANSI-C are also available in Handel-C. In addition, Handel-C adds some constructs that enable hardware design and direct mapping to FPGA. Figure 5.1 shows the shared constructs between ANSI-C and Handel-C and the added constructs by Handel-C.

In the following subsections, we discuss briefly some of the unique features of Handel-C. Most of these examples are extracted from [33]. Other examples are given in § 5.4 and § 5.5. More details can be found in [18, 19].

5.1.1.1 Timing

Each statement of Handel-C represents one clock cycle of execution.

Example 5.1 *The following code takes two clock cycles.*

```
a--;
if (a == 0)
    a++;
else
    delay;
```

The *delay* statement is used to do nothing for one clock cycle. It is inserted in the above code to balance execution time between the two branches.

[1] Handel-C is a trademark of Celoxica Limited.

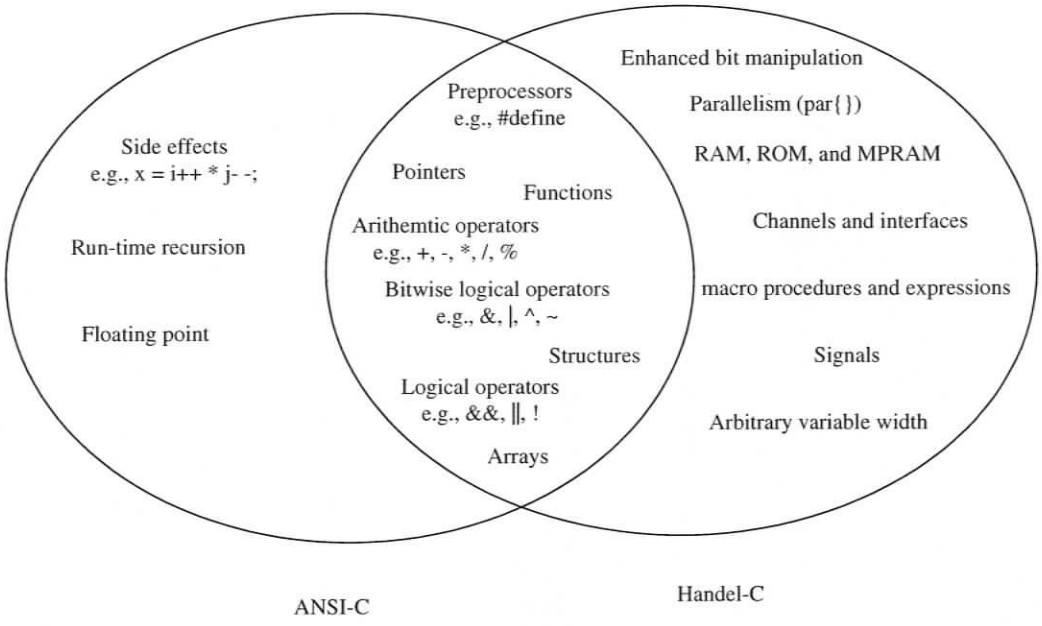

Figure 5.1. *Different and shared constructs between Handel-C and ANSI-C.*

5.1.1.2 Parallelism

The default in Handel-C is to execute statements sequentially. However, Handel-C allows parallelism by using the keyword *par*.

Example 5.2 *The following sequential block takes three clock cycles, whereas the parallel block takes only one clock cycle.*

```
// Sequential Block
seq
{
    a=1;
    b=2;
    c=3;
}
```

```
// Parallel Block
par
{
    a=1;
    b=2;
    c=3;
}
```

The keyword seq is optional in this case. If removed, the block will still be executed sequentially.

par and *seq* can be used to replicate a block of codes and execute the replicated copies in parallel or sequentially, respectively.

Example 5.3 *The following codes replicate the statement a[i] = b[i];*

```
seq(i=0; i<3; i++)
{
    a[i] = b[i];
}
```

```
par(i=0; i<3; i++)
{
    a[i] = b[i];
}
```

They are expanded as follows:

```
seq
{
    a[0] = b[0];
    a[1] = b[1];
    a[2] = b[2];
}
```

```
par
{
    a[0] = b[0];
    a[1] = b[1];
    a[2] = b[2];
}
```

5.1.1.3 Bit Manipulation

Handel-C enables enhanced bit manipulation, e.g., shift left and right, bit selection, and concatenation.

Example 5.4 *The following are examples of bit manipulation operations.*

(a) *Shift left and right:*

Let x be a 4-bit variable and y be an integer.

```
x = 0b1101; // initialize x to the binary value (1101)
y = 2;
x = x >> y; //shift x by two bits to the right, x becomes 0b0011
y--;
x = x << y; //shift x by one bit to the left, x becomes 0b0110
```

(b) *Take and drop bits:*

Let x be a 12-bit variable and y and z be two 4-bit variables.

```
x = 0xC27;  // initialize x to the hexadecimal value (C27)
y = x <- 4; // take the least significant 4 bits of x, y = 0x7
z = x \\ 8; // drop the least significant 8 bits of x, z = 0xC
```

(c) Bit selection and concatenation:
Let x be an 8-bit variable.

```
x = 0b11010011;  // initialize x to the binary value (11010011)
x = x[2:0] @ x[7:3]; // to rotate x to the left by 5 bits,
// concatenate the most significant 5 bits to the right of
// the least significant 2 bits, x becomes 0b01111010
```

5.1.1.4 Communication

The two constructs for communication in Handel-C are channels and interfaces. Channels are used for communication between parallel branches of code. Communication only occurs when both tasks are ready for the transfer which simplifies synchronization. Interfaces are used to communicate with external devices or external logic, such as other Handel-C programs or programs written in VHDL.

Example 5.5 *The following example shows how to send and receive a value through a channel.*

```
int 8 V; // declare an 8-bit variable V
chan int 8 Ch; // declare an 8-bit channel Ch
par
{
   Ch ! 23; // send the value 23 to the channel
   Ch ? V; // receive from channel and save in V
}
```

Example 5.6 *The following example shows how to receive and send values through interfaces.*

```
interface bus_in(int 4 I) InBus(); // declare an input bus InBus
   // with an input port I that receives a 4-bit value
interface bus_out() OutBus(int 4 O=y); // declare an output bus OutBus
   // with an output port O that sends a 4-bit value
int 4 x,y;
x = InBus.I; // receive from port I of interface InBus and save in x
```

```
y = x+1; // add one to x and send it to the output bus OutBus
         // (through the variable y)
```

5.1.2 DK Design Suite

DK design suite is an Integrated Development Environment (IDE) produced by Celoxica [77]. It provides a complete design flow from system specification all the way to hardware implementation. Using DK, one can compile, simulate, and debug a project written in Handel-C and synthesize it directly to FPGA chips.

DK includes a number of automatic optimization options that help a designer optimize his/her code before implementing it into hardware. These options include dead code elimination, common sub-expression elimination, and retiming [33]. It can also map the code to available ALUs on the target device. DK can also estimate device utilization and path delays using the Logic Estimator. It generates files that show the approximate area consumption of every line of the source code. It also generates files that show the lines of code that contribute to the longest path delay. This property allows the designer to optimize the code before synthesis.

Celoxica provides another tool called Platform Developer's Kit (PDK) [78]. This kit is composed of libraries and tools that support the DK design suite. PDK facilitates hardware independent simulation and development of components for specific hardware platforms. The main advantages of using PDK are reducing or removing the need for the developer to be concerned with hardware details, easy migration of designs between different platforms, and allowing design space exploration before final implementation decisions are made.

5.1.3 Xilinx ISE

To synthesize the designed unit and analyze its performance, we used the Xilinx ISE tools [79]. ISE consists of a number of tools that get HDL or EDIF files as input and generate an FPGA configurable bit files along with some post-place-and-route files that are

used for analyzing and optimizing the design. In particular, we used the Timing Analyzer to analyze timing and XPower to analyze power consumption.

Timing Analyzer [80] is used to analyze the timing of FPGA designs. It verifies that the delay on given paths of the design meets the timing constraint specified by the designer. It allows a designer to analyze the critical paths of the design, which are the paths that dictate the fastest time at which the design can run. The delay of the longest critical path is called *longest path delay*.

XPower [81] is used to analyze power consumption of an FPGA design. It generates reports that represent the total estimated power consumption of the design. This total power is the maximum power that the design will consume.

5.2 Design Flow

Figure 5.2 shows the design flow that we followed to design the proposed HMAC-hash unit. This design flow is a general one and may be used for any design [82]. In the following, we describe how we applied this design flow to our project:

1. The first step of the design flow is to translate the design specification into a Handel-C code.
2. The Handel-C code is compiled and simulated using the DK simulation tool for software verification. This step is described in § 5.3 in details.
3. During the simulation process, debugging and modifying the Handel-C code is performed whenever an error is detected.
4. As soon as the functionality is verified for correctness, as will be discussed in § 5.3, the Handel-C program is built into an EDIF netlist. The DK design suite provides this capability, which eliminates the need for HDL languages and increases flexibility.
5. Along with EDIF output, the DK Logic Estimator produces estimation files that assist in evaluating the designed project before generating the FPGA configuration file.

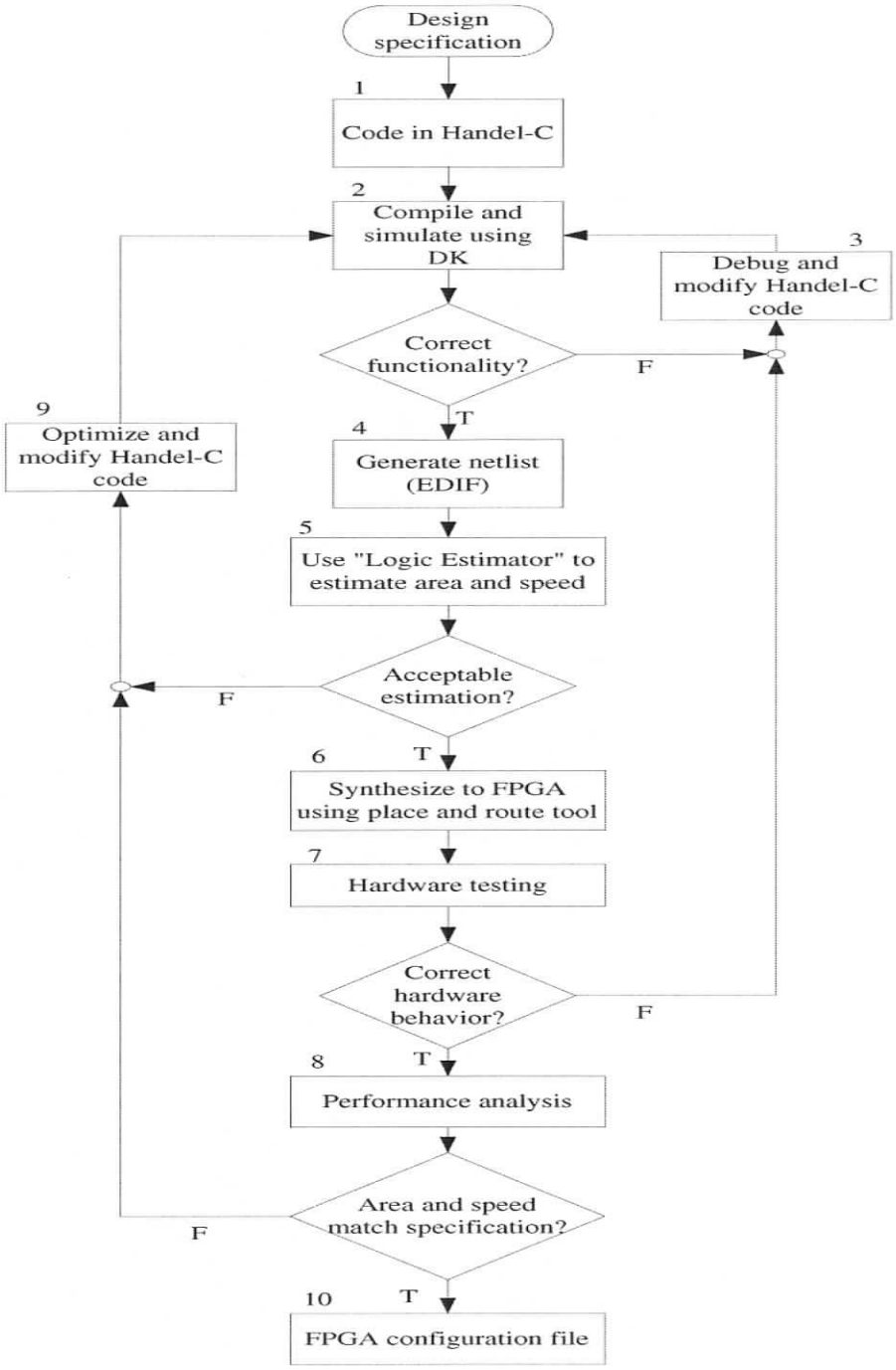

Figure 5.2. *Design flow followed in this dissertation.*

Numbers on the boxes correspond to steps explained in the text.

From these estimation files, we get estimated area and delay information rather than exact values. If the estimation results of both delay and area are acceptable, we proceed to the next step. Otherwise, further optimization is performed and we go to step 9.

6. The EDIF file is used by the Xilinx ISE place and route tool to synthesize the project into the FPGA chip. The place and route tool also generates some files that are used to analyze the performance of the designed project.
7. The synthesized unit is tested in hardware. If the hardware test passes, we go to step 8. If it does not pass, we go to step 3.
8. The performance of the designed unit is analyzed. This performance analysis is based on the files generated by the place and route tool. We consider four design factors for analysis: area, delay, throughput, and power consumption.
9. In this step, we may need to optimize the design. Optimization of the design is done by going back and modifying the Handel-C code and then repeating the above procedure. It should be noted that we only considered area and speed for optimization in this dissertation.
10. The final FPGA configuration file is generated. In our project, we synthesized the designed unit using the Xilinx ISE tool to a Virtex II FPGA chip (XC2V4000) [83].

5.3 Test Strategy

The following points illustrate the test strategy we followed:

1. **Functional test:** To verify the correctness of the functionality of the HMAC-hash unit, we used the DK simulation tools for software verification. First, we tested the hash unit using the test set vectors shown in Table 5.1, which we got from [4]. We used other working software available in [84, 85] to test other cases, such as different padding cases. Then, we tested the integrated HMAC-hash unit using the test vectors

Table 5.1. *Test vectors used to test the hash sub-unit.*

Test vector	Algorithm	Resulting hash value
""	MD5	0xd41d8cd98f00b204e9800998ecf8427e
	SHA-1	0xd41d8cd98f00b204e9800998ecf8427e
	RIPEMD-160	0xd41d8cd98f00b204e9800998ecf8427e
"a"	MD5	0x0cc175b9c0f1b6a831c399e269772661
	SHA-1	0x86f7e437faa5a7fce15d1ddcb9eaeaea377667b8
	RIPEMD-160	0x86f7e437faa5a7fce15d1ddcb9eaeaea377667b8
"abc"	MD5	0x900150983cd24fb0d6963f7d28e17f72
	SHA-1	0xc3fcd3d76192e4007dfb496cca67e13b
	RIPEMD-160	0x32d10c7b8cf96570ca04ce37f2a19d84240d3a89
"abcdefg hijklmnopq rstuvwxyz"	MD5	0xc3fcd3d76192e4007dfb496cca67e13b
	SHA-1	0x32d10c7b8cf96570ca04ce37f2a19d84240d3a89
	RIPEMD-160	0xf71c27109c692c1b56bbdceb5b9d2865b3708dbc

shown in Table 5.2. Some of these test vectors are specified in RFC's 2202 and 2286 to test HMAC with the three hash functions [86, 87]. We also added some extra test vectors to test other cases for message and key sizes in order to test all possible cases of key resizing and message padding. More test vectors can be found in [25, 27, 28, 30].

2. **Hardware test**: To test the HMAC-hash unit, we designed a testbench that sends some test vectors to the HMAC-hash unit, receives the result from it, and compares it with the correct result. If the result is correct, it sets an output signal to '1'. Otherwise, it sets that output signal to '0'. We used the test vectors shown in Table 5.2. If the hardware test does not pass, we go back to the debugging and modification step.

Table 5.2. *Test vectors used to test the integrated HMAC-hash unit.*

Message	**MS* (bits)**	**Key**	**KS* (bits)**	**Test case**
""	0	""	0	$MS = 0$, $KS = 0$
"what do ya want for nothing?"	224	"Jefe"	32	$MS \leq 447$, $KS \leq 447$
0xcd (repeated 50 times)	400	0x0102030405060708090 a0b0c0d0e0f1011121314 1516171819	200	$MS \leq 447$, $KS \leq 447$
"Test Using Larger Than Block-Size Key - Hash Key First"	432	0xaa (repeated 80 times)	640	$MS \leq 447$, $KS > 512$
"Test Using Larger Than Block-Size Key and Larger Than One Block-Size Data"	584	0xaa (repeated 80 times)	640	$MS > 512$, $KS > 512$
"abcdefghijklmnopqrs tuvwxyzabcdefghijkl mnopqrstuvwxyzabcd"	448	"abcdefghijklmnopqrs tuvwxyzabcdefghijkl mnopqrstuvwxyzabcd"	448	$447 < MS < 512$, $447 < KS < 512$
"abcdefghijklmnopqrs tuvwxyzabcdefghijkl mnopqrstuvwxyzabcd efghijkl"	512	"abcdefghijklmnopqrs tuvwxyzabcdefghijkl mnopqrstuvwxyzabcd efghijkl"	512	$MS = 512$, $KS = 512$

* MS is the message size and KS is the key size.

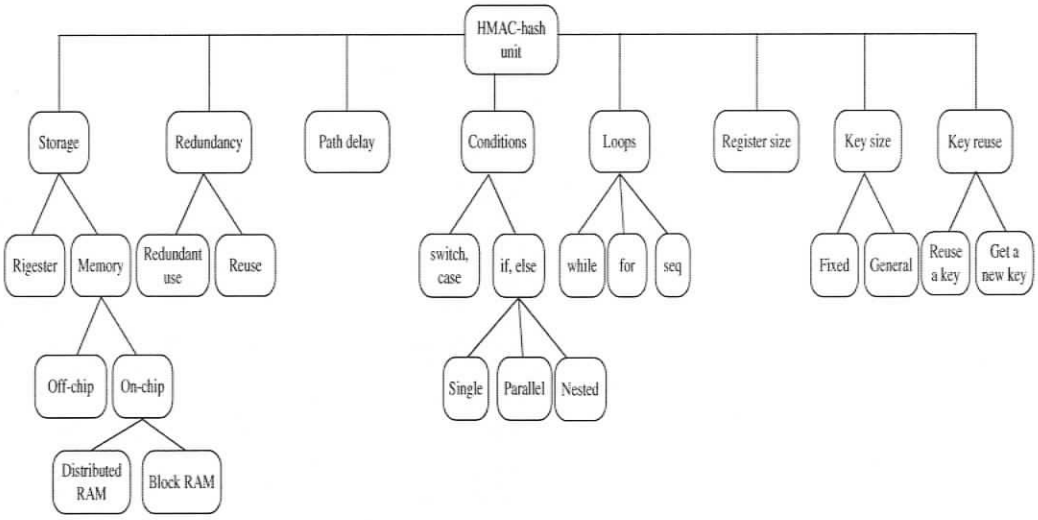

Figure 5.3. *Design space tree for the HMAC-hash unit.*

5.4 Design Space Exploration

For any hardware design, there are different implementation options, which determine its design space. Design space exploration intends to find out how a particular design option affects the performance of the design in order to have a trade-off between different performance metrics [88].

Figure 5.3 depicts the different design options that we considered for our HMAC-hash unit. Available Handel-C constructs enable conducting a design space exploration in order to get the most optimal results. In the following subsections, we discuss the design options that are implemented using different Handel-C constructs, which are all options shown in Figure 5.3 except the key size and key reuse. In § 4.2.3 and § 4.3.3, we discussed the key size and key reuse options, respectively.

5.4.1 Storage

Storage can be done in different ways. The default is to use registers. In Handel-C, this is done by declaring a variable or an array of variables. Similarly, declaring a constant or

an array of constants maps to registers in hardware. Registers can be accessed as much as needed in parallel and they are, in general, faster than memory. However, registers require more FPGA LUTs than memory.

Memory is an alternative option used for storage. Unlike registers, only one element of memory may be accessed in a single clock cycle. Multi-ported RAMs (MPRAMs) can be used to access more than one memory port in a single clock cycle. In Handel-C, RAM can be used instead of variables and ROM can be used instead of constants. The following example illustrates the difference between registers and memory.

Example 5.7 *Let X be an array of two 8-bit variables and Y be an array of two 8-bit constants. We have two options to define each of X and Y:*

(a) *For X:*

```
\\ Variable definition          \\ RAM definition
int 8 X[2];                     ram int 8 X[2];
```

(b) *For Y:*

```
\\ Constant definition          \\ ROM definition
const int 8 Y[2]={0,1};         rom int 8 Y[2]={0,1};
```

The following code is allowed for variables and constants but not allowed for RAM and ROM:

```
par(i=0;i<2;i++)
{
    X[i] = Y[i];
}
```

There are two options for memory, off-chip and on-chip. Because on-chip memory is generally faster than off-chip memory, we decided to use on-chip memory. For Xilinx Virtex II chips [83, 89], which we used in our design, there are two types of on-chip memory: block RAM and distributed RAM. Block RAM does not consume any LUTs, but it consumes a lot of memory cells. Distributed RAM does not consume memory cells, but consumes some LUTs. In addition, using block RAM leads to slower performance than using distributed RAM. In Handel-C, the *with* specification can be used to select one of the two options, as shown in the following code:

```
\\ Selecting distributed RAM
ram int 8 X[2] with {block = "SelectRAM"};
\\ Selecting block RAM
ram int 8 X[2] with {block = "BlockRAM"};
```

In order to have a balance between area and speed, we decided to use distributed RAM for variables that need to be accessed once in a clock cycle. For example, we used distributed RAM as ROM for the constants used for the algorithm initialization. If we need parallelism, we either use dual ported distributed RAM or registers. Dual ported distributed RAM is used when we need to access an element twice in parallel. As an example, the resized key K_0 of the HMAC algorithm is stored in a dual ported distributed RAM. Registers are used when we need more than two parallel accesses to the same element. The Alg signal is an example of a variable declared as a register.

5.4.2 Redundancy

The concept of redundancy applies to functions and registers. For functions, redundancy allows a function to be used more than once at a time, which helps with parallelism. However, this needs more area because each call to a function builds a replicated hardware copy of that function. On the other hand, reusing a function, which is called *shared function* in Handel-C, saves area. However, shared functions cannot be used more than once at a time [33].

The typical way of implementing redundant functions in Handel-C is to define a function and precede it with the keyword *inline* to have a replicated copy of the function whenever it is invoked. Another, but unusual, way is to distribute the required operation in all statements requiring this functionality instead of having one function of that operation and calling it from these statements. We noticed that sometimes, it is better with respect to area to have the required operation embedded in the same statement rather than defining a function, as shown in the following example:

Example 5.8 *Let X be a 32-bit word. To perform little-endian conversion on X, we tried*

two implementations:

(a) *We defined the following function:*

```
inline unsigned Little_Endian(unsigned 32 m)
{
    return(m[7:0] @ m[15:8] @ m[23:16] @ m[31:24]);
}
```

Then, we used this statement:

```
X = Little_Endian(X);
```

(b) *We used the following statement:*

```
X = X[7:0] @ X[15:8] @ X[23:16] @ X[31:24];
```

We found that the latter implementation option is better with respect to area.

Register reuse results in more multiplexed paths, which increases the delay in accessing the data stored in a certain register. This could be improved by introducing some redundancy. For registers, redundancy is done by defining more than one variable for the same data, which maps to multiple registers. Register redundancy decreases the delay required to access that data since it reduces the logic levels generated by multiplexing, but in general it increases the required area. However, redundancy could also decrease the area required as it might reduce the number of used multiplexors. For example, the Alg signal is used extensively in our code. In order to decrease the delay, we used three registers to store the value of Alg, and used each copy in different parts of the code. We noticed that this replication of the Alg register decreases both delay and area.

5.4.3 Path Delay

Path delay of an operation is the delay from the source to the destination of that operation. Splitting a certain operation into a number of sub-operations, each to be executed in one clock cycle usually decreases the path delay of the new sub-operations. This implies that the path delay is inversely proportional to the number of clock cycles required to execute an operation.

In general, this method of increasing clock cycles of an operation decreases the longest path delay of a design. However, care must be taken when throughput is considered. Throughput depends on both longest path delay, which determines the maximum clock frequency of a design, and the number of clock cycles required to process a certain task. So, the trade-off between the two parameters should be studied carefully.

Example 5.9 *The working variable A is updated in an SHA-1 compression step j using the following operation:*

$$A = ROTL^5(A) + f_j(B, C, D) + E + y_s[j] + X_j$$

This operation can be executed in one clock cycle or more, as follows:

(a) *Using one clock cycle:*

```
A = (A[26:0] @ A[31:27]) + f(B,C,D) + E + X[j] + y_s[j];
```

(b) *Using two clock cycles:*

```
par
{
    t1 = (A[26:0] @ A[31:27]) + f(B,C,D);
    t2 = E + X[j] + y_s[j];
}
A =  t1 + t2;
```

Since processing one 512-bit block using SHA-1 is done in 80 compression steps, the total number of clock cycles required to process a block is the result of multiplying 80 by the number of clock cycles required for one compression step. The more clock cycles for a compression step, the more clock cycles required for processing one 512-bit block, and that will affect the throughput as will be discussed in § 6.1.3. On the other hand, executing a compression step in one clock cycle may increase the longest path delay due to the complex operations required for a compression steps (addition, logical operations, left rotation, and accessing memory). Therefore, we found that using two clock cycles for each compression step is better for a balance between clock cycles and path delay, which results in a better throughput.

It should be noted that sometimes we found even if we increase the clock cycles of some complex operations, the longest path delay is still high. This is because the delay of the longest path in that case is due to the use of registers in different parts of the code which requires multi-levels of multiplexing in hardware. For example, using nested if conditions increases the path delay. In such cases, the longest path delay is not because of clock cycles of complex operations. Therefore, decreasing the clock cycles in this case is better.

5.4.4 Conditions

For conditions, we can use either if-$else$ or $switch$-$case$. The following example illustrates how these constructs can be used interchangeably.

Example 5.10 *To select between the three hash functions using the Alg signal, we can use one of the following:*

(a) *Using nested if-else:*

```
if(Alg == MD5)
    MD5_processing();
else if(Alg == SHA1)
    SHA1_processing();
else
    RIPEMD160_processing();
```

(b) *Using parallel if-else:*

```
par
{
    if(Alg == MD5)
        MD5_processing();
    else
        delay;
    if(Alg == SHA1)
         SHA1_processing();
    else
```

```
            delay;
        if(Alg == RIPEMD160)
            RIPEMD160_processing();
        else
            delay;
    }
```

(c) Using switch-case:

```
    switch(Alg)
    {
        case MD5:
        {
            MD5_processing();
            break;
        }
        case SHA1:
        {
            SHA1_processing();
            break;
        }
        case RIPEMD160:
        {
            RIPEMD160_processing();
            break;
        }
        default:
            break;
    }
```

For two options, such as checking the $Mode$ signal, we found that both *switch-case* and if-$else$ can be used interchangeably and have the same effects on both software and hardware. For multi options, one can select between *switch-case*, parallel if-$else$, or

nested if-$else$. In hardware, we found that $switch$-$case$ and parallel if-$else$ have the same performance. However, in cases that have complicated numerical conditions, it is more feasible from the implementation point of view to use if-$else$. An example of such cases is checking the padding case in the unified hash algorithm, which depends on the message size.

Nested if-$else$ has to be used with constructs that cannot be accessed in parallel, such as memory. Since nested if-$else$ results in more path delay than parallel if-$else$, the trade-off between area and speed must be studied carefully. Using registers allows one to use parallel if-$else$ statement, but it requires more area than using memory. On the other hand, usage of memory requires less area, but in this case one has to use nested if-$else$, which may increase the path delay.

5.4.5 Loops

Loops can be implemented in hardware in different ways. One way is to implement the loop iteration using a logic block and use this logic block for all iterations. This way needs some additional logic for checking the loop condition or checking the loop index. Another way is to implement each iteration in a different logic block, which needs more area for replicating the hardware logic.

In Handel-C, the constructs used for loops are $while$, for, and seq (which is used to replicate a piece of code a number of times and execute the replicated copies in sequence). With respect to area, $while$ loops use the same hardware logic for each iteration. In addition, they need some logic and storage for updating and checking the loop condition. Similarly, for loops use the same hardware logic for each iteration and need some storage for loop counters and end conditions and some logic for index increment and comparison. seq loops need more area because each loop iteration is implemented as a different logic block.

With respect to timing, for loops require one clock cycle for initializing the loop index and one extra clock cycle for incrementing or decrementing the index for each iteration.

These clock cycles can be saved using *while* or *seq* loops.

Example 5.11 *The compression steps of MD5 are executed 64 times, and each step takes two clock cycles. This loop can be implemented in the following ways:*

(a) *Using for loop:*

```
for(j=0; j!=64; j++)
{
    MD5_processing(j);
}
```

In this case, each iteration takes 3 clock cycles, two for MD5_processing() and one for incrementing j. In addition, one clock cycle is required to initialize j. So, the number of clock cycles required is $1 + 3 \times 64 = 193$.

(b) *Using while loop:*

```
while(j!=64)
{
    par
    {
        MD5_processing(j);
        j++;
    }
}
```

In this case, each iteration takes only two clock cycles. j can be initialized before the loop in parallel with some statements. Therefore, the number of clock cycles required is $2 \times 64 = 128$.

(c) *Using seq loop:*

```
seq(j=0; j!=64; j++)
{
    MD5_processing(j);
}
```

This requires the same number of clock cycles as the $while$ loop. However, the hardware logic used for an MD5 compression step is replicated 64 times.

From the above discussion, we can see that $while$ loops are generally better to use than for and seq loops. However, in some cases, we have to use seq loops, and that is when the loop index is used with constructs that require compile-time constants. Loop indexes used by seq loops are compile-time constants, whereas they are not compile-time constants for $while$ and for loops [33].

Example 5.12 *The resized key K_0 is XORed with $ipad$ and sent to the hash sub-unit (step 2 of Algorithm 2.1). Because the size of the Msg channel (see Figure 4.5) is 32, we need 16 clock cycles to send a 512-bit block. Since K_0 is stored as a 512-bit array, we need to index it 16 times, and in each time, we select 32 bits of K_0 and perform the XOR operation with 32 bits of $ipad$ and send the result to the hash sub-unit. Compile-time constants are required for bit selection, and hence, this example has to be done using seq loops, as shown in the following code:*

```
seq(y=0; y<=15; y++)
{
    Msg ! (K0.p1[0][511-(y*32):512-((y+1)*32)] ^ ipad32);
}
```

where ^ is the Handel-C XOR operator, and $ipad32$ is a 32-bit word of $ipad$.

In Handel-C, $while$ loops can be implemented using either the regular $while$ loops, or using $do\text{-}while$ loops. These two constructs have the same effects, with respect to both area and timing. The only difference occurs when the loop is embedded inside a condition (e.g., if condition or another $while$ loop). In that case, using $do\text{-}while$ is preferable in order to avoid combinational cycles caused by two consecutive conditions. It should be noted that $do\text{-}while$ can only be used if at least one iteration of the loop is to be executed [33, 90]. For example, word expansion is a step required for SHA-1 and it is executed 64 times. So, we first need to check the Alg signal, and if the hash function selected is SHA-1, then we will execute the loop 64 times. In this case, it is better to use $do\text{-}while$ than using $while$.

5.4.6 Register Size

Using larger size registers to store the message blocks decreases the number of clock cycles required to process them. However, using larger registers increases the area required. For example, we can use a 512-bit register to store a 512-bit block of a message. In this case, one clock cycle is required to receive that block, store it, and pass it to the hash sub-unit. Alternatively, we can use 16 32-bit registers to store one block. In this case, we need 16 clock cycles to receive and store a 512-bit block. The area required for the former case is much more than the latter case. The increase in area is not only for registers, but also for other logic blocks. In other words, using larger registers requires larger buses, adders, and so on.

5.5 Using Handel-C to Design the HMAC-Hash Unit

In the following subsections, we give highlights on how we used Handel-C to design and optimize the HMAC-hash unit.

5.5.1 General Skeleton

The following code is a general skeleton of the HMAC-hash unit using Handel-C. The comments in the code explain the role of each step.

```
macro proc Hash() // a procedure implementing the hash sub-unit
{
   initialization(); // initialize some chaining variables H1 to H5
       // using some initial chaining values C1 to C5, respectively.
   get_parameters();
     // get the message size, the hash function selected,
     // and compute n, which is the number of blocks in the message.
   while(i!=n) // i = 0 initially
   {
```

```
    preprocessing(i);
      // Perform preprocessing on the message block i:
      // receive the block, prepare it in the appropriate endian
      // format (big-endian for SHA-1 and little-endian for
      // MD5 and RIPEMD-160), and segment it into 16 32-bit
      // words X0 to X15.
    if(i==n-1)
      padding(); // perform padding on the last block
                 // (to make its size 512 bits).
    processing(i); // perform processing on block i.
  }
  completion(); // compute the final hash value.
}

macro proc HMAC() // a procedure implementing the HMAC sub-unit
{
  Get_Parameters();
  // get message and key sizes, hash function selected, and mode
  Key_Resize();
  // get the key and resize it according to its size. The result
  // is a 512-bit register K0.
  Get_massage(); // get the message M.
  if(mode==0) // HMAC is required
  {
      Send_to_Hash((K0 ^ ipad) @ M);
      // append the message M to the XOR of K0 and the constant ipad,
      // send it to the hash sub-unit, and get the result H.
      Send_to_Hash((K0 ^ opad) @ H);
      // append the H to the XOR of K0 and the constant opad,
      // send it to the hash sub-unit, and get the result MAC.
  }
  else // HMAC is not required
  {
      Send_to_Hash(M); // hash the message using
```

```
        // the selected hash function without HMAC
    }
    output(MAC); // output the final MAC value.
}


void main() // the main function
{
    par // run the hash and HMAC sub-units in parallel.
    {
        Hash();
        HMAC();
    }
}
```

5.5.2 Communication

The communication between the HMAC and hash sub-units is done using channels. The use of channels simplifies synchronization and handshaking.

The following code is used to declare the channels shown in Figure 4.5:

```
chan unsigned 32 Msg;
chan unsigned 32 Size;
chan unsigned 2 Sel;
chan unsigned 160 Hash;
chan unsigned 1 Hash_ready;
```

The first step of the hash sub-unit is to get the message size and the hash function selected. This step is done using the following code:

```
par {
    Hash_ready ! 1; // show that the hash sub-unit is ready to receive
    Size ? MsgSize; // read the message size (in bits) and
                    // store it in the local variable MsgSize
    Sel ? alg; // read the hash function to be used
               // and store in the local variable alg
```

}

After this step, the hash sub-unit gets the message, one 512-bit block at a time, from the Msg channel. Receiving one block requires 16 clock cycles. The hash sub-unit will not send another '1' on the $Hash\_ready$ channel until it finishes processing the current block. Hence, the HMAC unit will wait until the hash sub-unit is ready to receive another block. When the hash value is ready, the hash sub-unit sends it on the $Hash$ channel.

For the communication between the HMAC-hash unit and the outside world, we implemented and tested two mechanisms: channels and interfaces. Because channels are used for communication between parallel branches of code, they are used if the HMAC-hash unit and the communicating entity with it reside on the same FPGA chip. In this case, each entity is implemented using a different Handel-C code and run in parallel. This is done in a similar way as the communication between the HMAC and hash sub-units explained above. Using this option, synchronization is guaranteed.

Interfaces, on the other hand, are used if the entity that uses the HMAC-hash unit resides outside the FPGA chip on which the HMAC-hash is built. This is because interfaces are used to interface a Handel-C code to external devices and channels cannot be used in this case. One main difference between channels and interfaces is that interfaces do not imply synchronization as channels do. Instead, in each clock cycle, a new value is read from an interface. Therefore, synchronization must be taken into account when using interfaces. The input and output signals shown in Figure 4.5 are used for interfaces. Some of these signals are not needed if we use channels since synchronization is guaranteed using channels. For example, the MAC_ready signal is not needed because the HMAC sub-unit will not send the MAC through the channel until it is ready and valid. In other words, the communicating entity with the HMAC-hash unit will wait until a valid value is sent through the MAC channel, and after that, it will proceed to other statements.

5.5.3 Pipelining

In § 4.3.1, we discussed how we applied pipelining to the design of the HMAC-hash unit to increase its throughput. The following code is used to implement this pipeline:

```
preprocessing(i); // preprocessing of first block
if(n != 0) // do the following if # of blocks > 1
{
   while(i != n)
       par
       {
           processing(i);
           seq
           {
               seq(j=0; j!=32; j++)
               {
                   delay; // delay for 32 clock cycles to free the
                          // required registers
               }
               i++;
               preprocessing(i); // preprocessing of next block
               if(i == n-1)
                   padding();  // padding of last block
           }
       }
}
else
   padding(); // padding if # of blocks = 1
processing(i); //processing of the last block
```

where n is the number of 512-bit blocks in the message to be hashed (counting starts from 0), and i is initialized to '0'. After preprocessing the first block if there is more than one block, the processing stage starts processing the current block. During the processing of a block, the preprocessing stage starts preparing the next block. In order to do this, we used

another set of 16 32-bit registers, R_0 to R_{15}. The values of these registers are copied from X_0 to X_{15}, respectively. The *seq* block repeats the *delay* statement 32 times to make sure that the X registers are copied and become free (see § 5.5.4).

The padding process is done on the last block, which is the first block when $n = 0$. Finally, the last block is processed.

Since the preprocessing time is much shorter than the processing time, the preprocessing stage stalls waiting for the processing stage to finish processing the current block and start processing the next prepared block; then it starts preprocessing the next block, if there is any. This is guaranteed in the above code using *par*.

The initialization and completion stages shown in Figure 4.8 are not shown in the above code.

5.5.4 Parallelism

We used the *par* construct to reduce the time required to complete a task by utilizing some inherent parallelism in the unified hash algorithm. This utilization has been discussed in § 4.3.2. The following codes illustrate how we used Handel-C to apply parallelism to the design of the HMAC-hash unit (see the general skeleton code in § 5.5.1):

(a) Running the initialization step in parallel with the preprocessing step (Figure 4.9 (a)):

```
par
{
   initialization();
   preprocessing();
}
```

(b) RIPEMD-160 has two processing lines, a right processing line and a left processing line. In our unified hash algorithm, the left processing line of RIPEMD-160 is embedded in the unified processing line. The two processing lines (unified and RIPEMD-160 right processing line) can run in parallel (Figure 4.9 (b)).

(c) Running the expansion process of SHA-1 in parallel with the unified processing line (Figure 4.9 (c)).

The following code illustrates cases (b) and (c):

```
par
{
  if(alg==RIPEMD160)
    RIPEMD_right_processing();
  else
    delay;

  if(alg==SHA1)
  {
    do
    {
       R[i] = X[i]; // i is initialized to 0
       i++;
    }  while(i != 16);
    do
    {
     R[i<-4]=R[(i<-4)-3] ^ R[(i<-4)-8] ^ R[(i<-4)-14] ^ R[(i<-4)-16];
     par
     {
       R[i<-4]=R[i<-4][30:0] @ R[i<-4][31];
       i++;
     }
    } while(i != 80);
  }
  else
   delay;

// code for the unified processing
 }
```

where ^ is the Handel-C XOR operator. It should be noted that the word expansion takes two clock cycles because the SHA-1 processing takes two clock cycles in our implementation. If one clock cycle is targeted, the word expansion must be done in one statement.

5.5.5 Reconfigurability

The HMAC sub-unit is reconfigured by 2 inputs: $Mode$ and Alg as shown in Table 4.4 (see § 4.2.2). Reconfigurability is done in Handel-C using either $if\text{-}else$ or $switch\text{-}case$, as explained in § 5.4.4. In the general skeleton code shown in § 5.5.1, $if\text{-}else$ is used. The following code shows how $switch\text{-}case$ can be used instead:

```
switch(mode)
{
   case 0: // HMAC is required
   {
       Send_to_Hash((K0 ^ ipad) @ M);
       Send_to_Hash((K0 ^ opad) @ H);
       break;
   }
   case 1: // HMAC is not required
   {
       Send_to_Hash(M);
       break;
   }
   default:
       break;
}
```

5.6 Concluding Remarks

In this chapter, we discussed the experimental work and the design methodology we followed to design the proposed unit. This work is the first in the literature that uses this methodology to implement hash functions and HMAC. This methodology increases the

level of flexibility by using high level languages to directly map the design to FPGA platforms. At the same time, it increases the performance by using reconfigurable hardware. The application of this methodology to design our proposed HMAC-hash unit has been submitted for publication in [91, 92].

The use of Handel-C to design the proposed unit allows us to conduct a design space exploration in order to get the most optimal performance. We discussed some examples of design factors used for this purpose. The design space exploration discussed in this chapter has been submitted for publication in [93].

In the following chapter, the performance of the proposed unit is analyzed. This analysis includes four factors: area, delay, throughput, and power consumption. The results obtained are compared to results reported in previous works.

Chapter 6

Performance Analysis and Comparison

In this chapter, the performance of the proposed unit is analyzed in § 6.1. Our study considers four design factors: area, delay, throughput, and power consumption. We also discuss the effect of the enhancement techniques we applied on the performance of the designed unit. In § 6.2, the results obtained by the proposed unit are compared to results reported in previous work. First, we compare our work to those integrating more than one hash function on FPGA without integrating HMAC. Then, we compare our work to those integrating HMAC with some hash functions.

6.1 Performance Analysis

The performance of our proposed design is analyzed for the four implementations described in § 4.3.3, which are:

1. General key size with no key reuse, denoted with $D1$.
2. General key size with key reuse, denoted with $D2$.
3. Fixed key size with no key reuse, denoted with $D3$.
4. Fixed key size with key reuse, denoted with $D4$.

6.1.1 Area Analysis

Area is measured by the amount of hardware resources occupied by the designed HMAC-hash unit. Table 6.1 shows detailed figures comparing area requirements for the four implementations.

Table 6.1. *Details of area requirements of the HMAC-hash unit.*

Implementation*	**D1**	**D2**	**D3**	**D4**
Total number of used LUTs** (Available = 46,080)	14,911	14,913	12,962	12,970
LUTs utilization	32.36%	32.36%	28.13%	28.15%
LUTs used as logic	12,740	12,742	11,623	11,631
LUTs used as a route-thru	647	647	519	519
LUTs used for dual ported RAMs	1,216	1,216	512	512
LUTs used as 16x1 RAMs	32	32	32	32
LUTs used as 16x1 ROMs	263	263	263	263
LUTs used as shift registers	13	13	13	13
Total number of flip flops (Available = 46,080)	3,540	3,552	3,422	3,431
Flip flops utilization	7.68%	7.71%	7.43%	7.45%
Number of bonded IOBs (Available = 824)	169	170	137	138
IOB utilization	20.51%	20.63%	16.63%	16.75%

* D1 = the general implementation of HMAC.

D2 = the general implementation of HMAC with key reuse.

D3 = the HMAC implementation with fixed key size.

D4 = the HMAC implementation with fixed key size and key reuse.

** Total number of used LUTs includes LUTs used as logic, memory, routing, and shift registers.

The table shows clearly that implementations with fixed key size (D3 and D4) consume less area than implementations with general key size (D1 and D2). The savings in area

occurs in LUTs used as logic, dual ported RAM, and routing, and in flip flops. In addition, the number of IOBs (input/output blocks) of a fixed key size implementation is less than the corresponding general implementation by 32, which is the number of input ports used for key size.

We can notice from the table that the proposed key reuse mechanism (D2 and D4) does not add much area to the implementations that do not include this feature (D1 and D3). It adds few LUTs, few more flip flops, and one IOB (for the $Same\_key$ signal). However, it increases the throughput for reused keys, especially when key sizes are long, as will be discussed in § 6.1.3. Therefore, we recommend including this feature in any HMAC implementation.

In order to show the effectiveness of the proposed HMAC-hash unit, we implemented each algorithm of the six considered alone. Table 6.2 shows the required area of each hash function implemented separately. We can see from Tables 6.1 and 6.2 that our proposed unified architecture (using D1) requires only 37.87% of the sum of the area required by the six hash functions if each one is implemented alone. This shows the goodness of the proposed HMAC-Hash unit.

Table 6.2. *Required area of each hash function implemented separately.*

Hash function	MD5	SHA-1	RIPEMD-160	HMAC-MD5	HMAC-SHA-1	HMAC-RIPEMD-160	Total
Area (LUTs)	4,353	3,903	6,072	7,911	7,484	9,650	39,373

It should be noted that we used the same specifications of the HMAC-hash unit in the implementations of each hash function. This includes timing constraints, pipelining, parallelism, and logic blocks and memory cells required for all parts (e.g., HMAC, initialization, preprocessing, processing, word expansion, and completion).

6.1.2 Delay Analysis

Table 6.3 shows the details of the longest path delay of the proposed unit. The total delay consists of two parts:

(a) **Data path delay**: which is the delay from a source to a destination. This delay is the sum of two delays: logic delay and routing delay.

(b) **Clock path skew**: which is the difference between the time a clock edge arrives at the source of a path and the time it arrives at the destination of that path.

Table 6.3. *Details of the longest path delay of the HMAC-hash unit.*

Implementation*	**D1**	**D2**	**D3**	**D4**
Delay (ns)	23.005	23.144	22.551	22.680
Data path delay (ns)	23.005	23.097	22.551	22.604
Clock path skew (ns)	0	-0.047	0	-0.076
Logic delay (ns)	11.23	13.645	12.050	13.366
Percentage of logic delay to data path delay	48.8%	59.1%	53.4%	59.1%
Routing delay (ns)	11.775	9.452	10.501	9.238
Percentage of routing delay to data path delay	51.2%	40.9%	46.6%	40.9%

* D1 = the general implementation of HMAC.

D2 = the general implementation of HMAC with key reuse.

D3 = the HMAC implementation with fixed key size.

D4 = the HMAC implementation with fixed key size and key reuse.

The added delay when including the key reuse mechanism (D2 and D4) is small (less than 0.15 ns). However, the saving in clock cycles using this mechanism is considerable for reused large keys. Therefore, the overall throughput of HMAC using this feature will be better, as will be shown in § 6.1.3.

On the other hand, the key resizing step, which is used for the general HMAC implementations (D1 and D2), adds more delay than key reuse (about 0.4 ns). Therefore,

implementing HMAC with fixed key size is preferable as long as security issues regarding key selection are taken into account.

We can see from the table that the routing delay is a very important factor in the total delay. The minimum of routing delay is 40.9%. In our implementation, we relied on automatic routing. Manual routing may be used for further optimization.

6.1.3 Throughput Analysis

Throughput is defined as the number of messages processed in a unit time. To be more accurate, we measure throughput in bps. The general equation to compute throughput is

$$Throughput = \frac{MS}{T_M} \tag{6.1}$$

where MS is the number of bits of a message M and T_M is the time required to process the message M, which is computed as follows:

$$T_M = T_c \times CC_M = \frac{CC_M}{F} \tag{6.2}$$

where T_c is the clock period, CC_M is the number of clock cycles required to process the message M, and F is the maximum clock frequency ($F = \frac{1}{T_c}$).

So, the throughput of processing one message block using hash functions is computed according to the following equation [73]:

$$Throughput_{block} = \frac{BS \times F}{CC_b} \tag{6.3}$$

where BS is the number of bits of the block (512 bits in our case), and CC_b is the number of clock cycles required to process one block.

The overall throughput of hashing a message is given by the following equation:

$$Throughput_{hash} = \frac{N \times BS \times F}{CC_{pc} + (N \times CC_b)} \tag{6.4}$$

where CC_{pc} is the number of clock cycles required for preprocessing and completion of hashing a message, and N is the number of message blocks. When N is large enough, we

can ignore the term CC_{pc}, and Equation 6.4 will be equal to Equation 6.3:

$$Throughput_{hash} = Throughput_{block} = \frac{BS \times F}{CC_b} \tag{6.5}$$

It is difficult to find an exact figure for the throughput of the general HMAC algorithm because it depends on two parameters, key size and message size. To find out the throughput of the HMAC-hash unit, we use the following equation:

$$Throughput_{HMAC} = \frac{N \times BS \times F}{3 \times CC_{pc} + (3 + N + k)(CC_b)} \tag{6.6}$$

where k is the number of key blocks if its size is more than 512. If its size is less than or equal to 512, it is resized without hashing, and in this case, k equals zero in Equation 6.6. In the HMAC algorithm (Algorithm 2.1 in § 2.3.1), there are three additional blocks need to be hashed. Therefore, we multiply $(3+N+k)$ by CC_b. We multiply CC_{pc} by 3 because HMAC uses the hash function three times.

When N is large enough, we can ignore the term $3 \times CC_{pc}$, and Equation 6.6 can be rewritten as follows [73]:

$$Throughput_{HMAC} = \frac{N}{3 + N + k} \times Throughput_{hash} \tag{6.7}$$

Table 6.4 shows the experimental information required to compute the throughput of the four implementations.

The clock cycles required for the HMAC algorithm are computed per message. This number of clock cycles includes the cycles required to receive the key, prepare the message (and key if required) for hashing, and output the MAC. However, it does not include the time required for hashing, which is included in the next rows of the table. For the implementations with general key size, the HMAC algorithm takes from 7 to 24 clock cycles, depending on the key size and on the $Same\_key$ signal (if used). For the implementations with fixed key size, there is one extra clock cycle required for receiving the key when the selected algorithm is HMAC-SHA-1 or HMAC-RIPEMD-160.

The clock cycles required for hash preprocessing and completion are also computed per message. The preprocessing time required for hashing includes receiving a message block,

Table 6.4. *Different factors to compute the throughput of the HMAC-hash unit.*

Implementation*	**D1**	**D2**	**D3**	**D4**
Maximum frequency (MHz)	43.47	43.21	44.34	44.1
Clock cycles for the HMAC algorithm	10-24	7-24	11-12	7-12
Clock cycles for hash preprocessing and completion for MD5	23-26			
Clock cycles for hash preprocessing and completion for SHA-1	20-23			
Clock cycles for hash preprocessing and completion for RIPEMD-160	23-26			
Clock cycles for processing one 512-bit block for MD5	130			
Clock cycles for processing one 512-bit block for SHA-1	162			
Clock cycles for processing one 512-bit block for RIPEMD-160	162			
Average MD5 throughput of a 512-bit block (Mbps) (Equation 6.3)	171.2	170.2	174.63	173.69
Average SHA-1 throughput of a 512-bit block (Mbps) (Equation 6.3)	137.4	136.56	140.14	139.38
Average RIPEMD-160 throughput of a 512-bit block (Mbps) (Equation 6.3)	137.4	136.56	140.14	139.38

* D1 = the general implementation of HMAC.

D2 = the general implementation of HMAC with key reuse.

D3 = the HMAC implementation with fixed key size.

D4 = the HMAC implementation with fixed key size and key reuse.

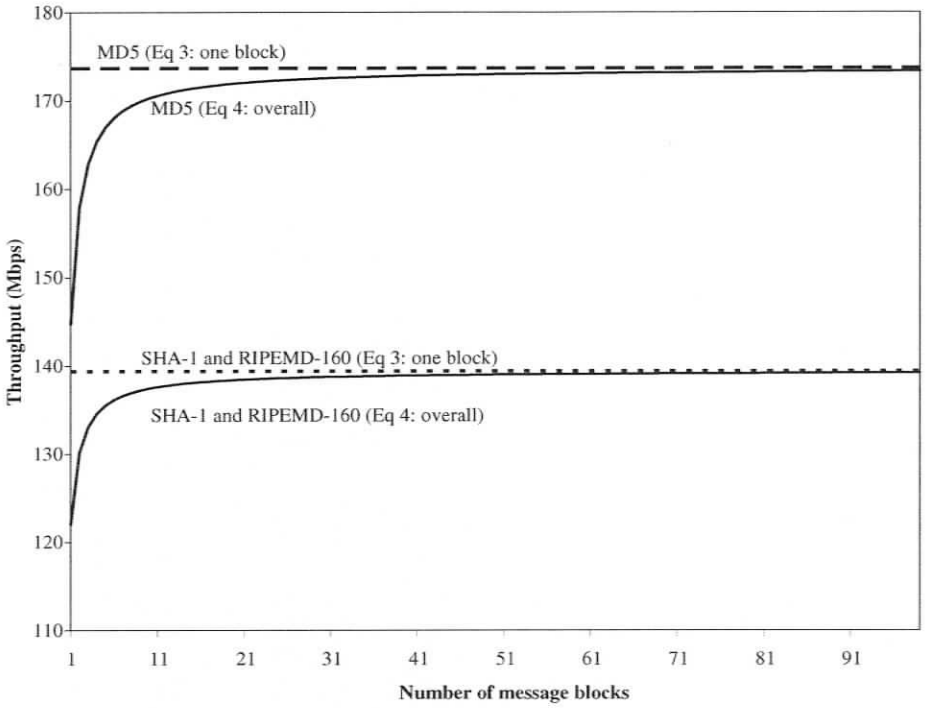

Figure 6.1. *Overall hash throughput vs. throughput of one message block.*
Results are shown for D4 only for illustrating purposes.

segmenting it into 16 32-bit blocks, preparing it in the appropriate endian format, and padding the last message block. The completion time includes concatenating the chaining variables to form the hash value, preparing it in the appropriate endian format, and sending it to the HMAC sub-unit. The number of clock cycles required for all these steps is small compared to the number of cycles required for processing one 512-bit block. Therefore, the larger the message size, the closer the overall throughput to the throughput of processing one 512-bit block. The number of clock cycles required for processing a message block includes a clock cycle for initializing the working variables, a clock cycle for updating the chaining variables, and two clock cycles per round (64 rounds for MD5 and 80 rounds for each of SHA-1 and RIPEMD-160).

Figure 6.1 draws Equations 6.3 and 6.4 for $1 \leq N \leq 100$. It shows that Equation 6.4 approaches Equation 6.3 for larger message sizes, which means that Equation 6.5 is true for large N.

The best case for $Throughput_{HMAC}$ occurs when the key size is less than 512 and N is large, and in that case the HMAC throughput is almost the same as the hashing throughput. The worst case occurs when k is large compared to N, and hence the throughput will be very small. Between these two extremes, HMAC throughput will be a fraction of the hashing throughput, depending on N and k.

Figure 6.2 depicts the results of Equations 6.3, 6.6, and 6.7 for key sizes less than or equal to 512 and $1 \leq N \leq 1024$. The figure illustrates the following (for large N):

1. Equation 6.6 can be approximated using Equation 6.7.
2. The larger the message size, the closer the overall HMAC throughput to the throughput of processing one message block.

These two facts are true for the general HMAC implementations when the key size is less than or equal to 512, and also for the HMAC implementations with fixed key size.

For key sizes greater than 512, key reuse increases the HMAC throughput for reused keys by eliminating the k term from Equation 6.7, which approaches the throughput of processing one message block for larger message sizes.

6.1.4 Power Consumption Analysis

Table 6.5 shows the estimated power consumption of the four implementations. We include this table to have a complete picture of the proposed design, although we did not optimize the design for power consumption. The total estimated power consists of two parts:

(a) **Dynamic power:** which is the power consumed by switching activities. This includes clock, input, output, logic, and internal signals.

(b) **Quiescent power:** which is the power consumed with no signal switching. It is also called *static power*.

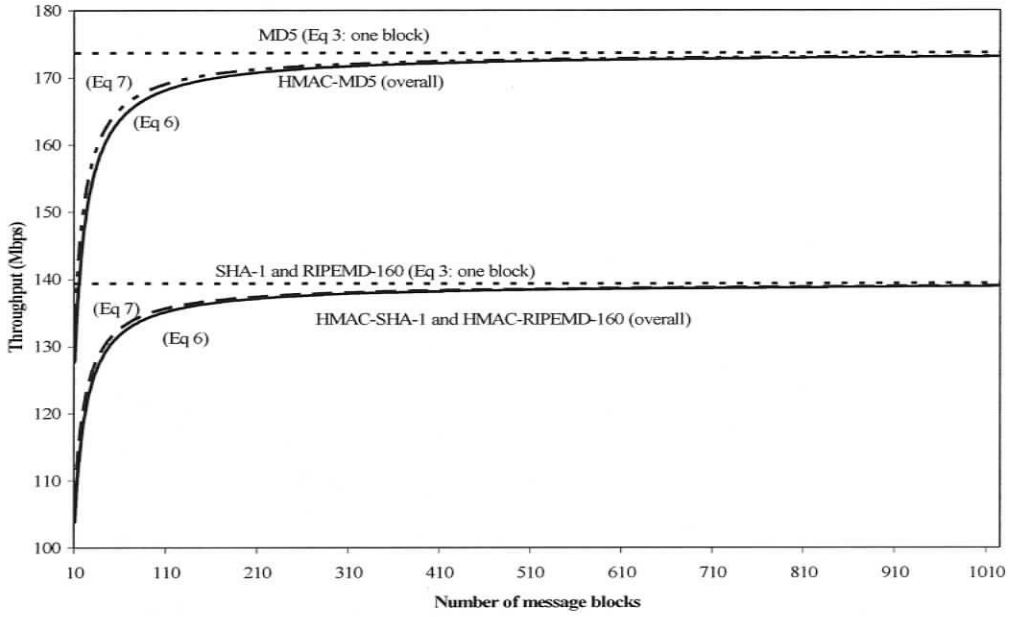

Figure 6.2. *Overall HMAC throughput vs. throughput of one message block.*
Results are shown for D4 only for illustrating purposes.

Table 6.5. *Details of the estimated power consumption of the HMAC-hash unit.*

Implementation*	**D1**	**D2**	**D3**	**D4**
Total power consumption (mW)	876	867	805	793
Dynamic power (mW)	201	192	130	118
Quiescent power (mW)	675	675	675	675

* D1 = the general implementation of HMAC.

D2 = the general implementation of HMAC with key reuse.

D3 = the HMAC implementation with fixed key size.

D4 = the HMAC implementation with fixed key size and key reuse.

The dynamic power shown in the table includes only clock and logic signals, because the other types of signals depend heavily on the external loading capacitance of the de-

signed unit. We notice from the table that the quiescent power is the same for all implementations. On the other hand, the dynamic power increases with the increase in area as well as in clock frequency. D1 and D2 consume more area than D3 and D4 (see Table 6.1), and hence D1 and D2 consume more power than D3 and D4. On the other hand, D1 has more frequency than D2 (see Table 6.4), and therefore, D1 consumes more power than D2, although D1 is consumes slightly less area than D2 (2 LUTs). Similarly, D3 consumes more power than D4 because it has higher frequency, although it consumes less area (8 LUTs). These results show that frequency has more impact on dynamic power than area, to a certain level.

To optimize the design for power consumption, we have two options. The first is to decrease the area required by removing some functionality from the design. The second is to decrease the frequency, and in this case, there will be a trade-off between power consumption and throughput.

6.1.5 Effect of the Design Enhancements on the Performance

In order to show the effect of the enhancement techniques that we applied to improve the designed unit, as discussed in § 4.3, we examine in this subsection the performance of the designed unit before and after applying these enhancements.

Table 6.6 summarizes our findings from the performance analysis discuss in the previous subsections. The table shows that the proposed key reuse mechanism improves the HMAC throughput significantly when a large key is reused, with negligible increase in area and delay. It also shows that the implementation of HMAC for fixed key size is better in area, delay, throughput, and power consumption. Therefore, it is better to implement the HMAC algorithm for fixed key size as long as security issues regarding key selection are taken into account, and in this case, it is better not to include the key reuse mechanism. On the other hand, we recommend including the key reuse mechanism in any implementation of the HMAC algorithm with general key size.

Table 6.7 shows the performance of the HMAC-hash unit before and after applying

Table 6.6. *Summary of the performance analysis of the HMAC-hash unit.*

Design metric	**Conclusion**
Area	* Implementations with fixed key size consume less area than implementations with general key size. * Including the key reuse mechanism does not add much area to the implementations that do not include this feature.
Delay	* Implementations with fixed key size have less delay than implementations with general key size. * Including the key reuse mechanism does not add much delay to the implementations that do not include this feature. *Routing delay is an important factor of total path delay.
Throughput	* The hash throughput is almost equal to the throughput of processing one message block. * HMAC with fixed key size has a throughput almost equal to the throughput of processing one message block for large message sizes. * The key reuse mechanism makes the HMAC throughput almost equal to the throughput of processing one message block for large message sizes.
Power consumption	* HMAC with fixed key size and key reuse consumes the least power. * Frequency has more impact on power consumption than area.

the enhancements. We used the results of D2 because it includes all design enhancements (pipelining, parallelism, and key reuse with general key size). The area of the design before enhancement is a bit less than the area of the enhanced design. This is expected since the enhancements require more area resources. On the other hand, we can see from the table that the delay and the clock cycles of the design before enhancement is more than the enhanced one, which both affect the throughput. The average throughput of processing one 512-bit block in the design before enhancement is computed using Equation 6.3. In this case, CC_b is the sum of the clock cycles required for preprocessing and processing, because preprocessing is done for each block. In the enhanced design, the preprocessing clock cycles are computed per message because of the pipeline that we applied. In addition, we notice from the table that the clock cycles required for processing one 512-bit block for SHA-1 and RIPEMD-160 in the design before enhancement is much more than the clock cycles required by the enhanced design because the expansion process of SHA-1 and the right processing line of RIPEMD-160 are run sequentially with the unified processing line.

For HMAC, the proposed key reuse mechanism increases the throughput when a large key is used in successive messages. This is not the case for the design before enhancement. Therefore, the HMAC throughput will be worse for large keys.

6.2 Comparison with Previous Work

In this section, we compare the results obtained using our proposed reconfigurable HMAC-hash unit with those reported in previous work. First, we compare our work with those integrating more than one hash function without HMAC. Then, we compare our work with those integrating the HMAC algorithm with some hash functions.

Table 6.7. *Comparing the performance of the HMAC-hash unit before and after enhancement.*

Results	**Before enhancement**	**After enhancement***
Total number of used LUTs	14,019	14,913
Maximum frequency (MHz)	42.77	43.21
Clock cycles for the HMAC algorithm	10-24	7-24
Clock cycles for hash initialization and completion	3-6	2-5
Clock cycles for hash preprocessing	18-21	18-21
Clock cycles for processing one 512-bit block for MD5	130	130
Clock cycles for processing one 512-bit block for SHA-1	290	162
Clock cycles for processing one 512-bit block for RIPEMD-160	322	162
Average MD5 throughput of a 512-bit block (Mbps)	147.96	170.2
Average SHA-1 throughput of a 512-bit block (Mbps)	71.1	136.56
Average RIPEMD-160 throughput of a 512-bit block (Mbps)	64.41	136.56

* We used the results of D2 as it uses the key reuse mechanism.

6.2.1 Comparison with Work that Integrated More than One Hash Function

In this subsection, we compare our designed unit with work integrating more than one hash algorithm. We only consider work that integrated more than one hash algorithm on FPGA and we only include results for the three hash functions MD5, SHA-1, and RIPEMD-160. Table 6.8 summarizes the comparison between our work and previous work.

All listed work reported the area using building blocks of the specified FPGA chip. However, building blocks of different FPGA chips may not have the same number of LUTs.

Table 6.8. *Comparison with work that integrated more than one hash function without HMAC.*

Designs and proposals	**Ng [65]**	**Wang [73]**	**Kang [47]**	**Dominikus [70]**	**Proposed design♣**
Hash functions implemented*	M, R	M, S	M, S♠	M, S, R	M, S, R
FPGA vendor	Altera	Altera	Altera	Xilinx	Xilinx
FPGA chip	EPF10 K50	EP20 K1000	EP20 K1000	XC V300E	XC2 V4000
Area cost (LUTs)	1,964	3,040	10,573	4,493	12,970
LUTs used for the processing block	1964	3040	×	4493	7,074
Maximum frequency (MHz)	26.66	22.67	18.0	42.9	44.1
Clock cycles for MD5**	66	65	65	206	130
Average MD5 throughput (Mbps)**	206	178.6	142	107	173.69
Clock cycles for SHA-1**	-	81	81	255	162
Average SHA-1 throughput (Mbps)**	-	143.3	114	86	139.38
Clock cycles for RIPEMD-160**	162	-	-	337	162
Average RIPEMD-160 throughput (Mbps)**	84	-	-	65	139.38

* We only include results for the three hash functions MD5 (M), SHA-1 (S), and RIPEMD-160 (R)

** Results are shown per one 512-bit block.

♠ HAS-160 (The Hash function Algorithm Standard) is integrated with SHA-1 and MD5.

♣ Results are shown for D4.

"-" means the corresponding hash function is not implemented.

"×" means the corresponding information is not reported.

Therefore, for each work, we referred to the data sheet of the FPGA chip used and found out the number of LUTs equivalent to the reported result in order to unify the area units for comparison. Most of the listed work showed results only for the processing block. However, we include in our results the area cost of the HMAC sub-unit (using the fixed key size and key reuse implementation, D4) along with all required blocks of the hash sub-unit (preprocessing, processing, and completion blocks), including area used for logic, memory, routing, and shift registers.

For a better comparison, we include also the area required for the processing block of our work in comparison with other work. Kang et al. [47] did not include information for the processing block. Ng et al. [65] implemented only one processing line of RIPEMD-160 and did not include SHA-1 in their work. Wang et al. [73] did not include RIPEMD-160, which is the most demanding algorithm among the three considered with respect to area requirement. Dominikus [70] only included the area of the compression step, which excludes other parts of the processing block such as chaining variable updates. In addition, all these works (Ng. et al. [47], Wang et al. [73], and Dominikus [70]) did not include the RAM required to store the constants, the original and expanded message words, and the chaining and working variables in the area reported. This justifies the relatively higher area cost that we are reporting as compared to others, since we include all parts required for the processing block (e.g., RAM for all constants, variables, and message words, and area required for pipelining and parallelism). In addition, having a unified engine for six standardized algorithms (the three hash algorithms with and without the HMAC algorithm) is still better in area than having a module for each one of them, as shown in Table 6.2.

On the other hand, when we look at the throughput, we find that our work outperforms those integrating three or more algorithms (Kang et al. [47] and Dominikus et al. [70]). We also achieve better throughput than Ng et al. [65] for RIPEMD-160, but less for MD5. However, SHA-1 is not included in that work. Although the throughput achieved by our work is slightly less than that achieved by Wang et al. [73] for both MD5 and SHA-1, that work did not include RIPEMD-160, which is more complicated in terms of area and

processing requirements.

Furthermore, we achieve better maximum frequency than all other work. Some of the work, such as Dominikus [70], proposed to increase the number of clock cycles required to complete one processing step in order to increase the maximum frequency. Even though, we still achieve better maximum frequency than these works without having to increase the number of clock cycles required for the processing step as compared to them. It should be noted here that keeping the required number of clock cycles as small as possible helped us achieve a higher throughput for one processing step. Additionally, utilizing pipelining and parallelism in our unified architecture increased the overall throughput.

6.2.2 Comparison with Work that Integrated HMAC with Some Hash Functions

Most of the work in the literature focus on HMAC-SHA-1 implementations (Selimis et al. [71], Michail et al. [72], and McLoone and McCanny [49]). Lu and Lockwood [50] implemented HMAC-MD5 and HMAC-SHA-1 separately. Wang et al. [73] integrated HMAC with MD5 and SHA-1. However, there is no work that addressed FPGA design of HMAC-RIPEMD-160.

Table 6.9 shows the results of these designs compared to our achieved results. As we mentioned before, it is difficult to find out an exact figure of HMAC throughput. Selimis et al. [71] did not address the throughput of HMAC. They only addressed the throughput of processing one 512-bit block using SHA-1, which is 518 Mbps. On the other hand, McLoone et al. [49] estimated the HMAC throughput by computing the number of clock cycles required. They reported that a throughput of 78 Mbps with a clock rate of 50 MHz is achieved. However, since they designed the HMAC-SHA-1 unit to be used in parallel with an encryption core, the operating rate is 24.2 MHz for both cores, which results in a maximum throughput of 37.8 Mbps. Michail et al. [72] implemented HMAC-SHA-1 in a pipelined architecture. The number of clock cycles required to obtain an HMAC result

Table 6.9. *Comparison with work that integrated some hash functions with HMAC.*

Designs and proposals	**Selimis [71]**	**Wang [73]**	**McLoone [49]**	**Michail [72]**	**Lu [50]**		**Proposed design♡**
FPGA vendor	Xilinx	Altera	Xilinx	Xilinx	Xilinx		Xilinx
FPGA chip	XC V50	EP20 K1000	XC V1000E	XC V3200E	XC2 VP100		XC2 V4000
Hash functions implemented*	S	M, S	S	S	M	S	M, S, R
Area cost (LUTs)	1,593	5,329	14,494◇	12,022	4,050♠	5,977♠	14,913
Maximum frequency (MHz)	82	21.96	24.2	62.0	106.6	145.3	43.21
Average HMAC-MD5 throughput (Mbps)	-	43.2	-	-	277.1♣	-	170.2
Average HMAC-SHA-1 throughput (Mbps)	518♣	34.7	37.8	1587	-	303.6♣	136.56
Average HMAC-RIPEMD-160 throughput (Mbps)	-	-	-	-	-	-	136.56

* We only include results for HMAC with the three hash functions:
MD5 (M), SHA-1 (S), and RIPEMD-160 (R)

◇ This number includes area cost of the HMAC-SHA-1 core and an encryption core.

♠ Results are shown for the hash function core.

♣ Results are shown per one 512-bit block.

♡ Results are shown for D2.

"-" means the corresponding hash function is not implemented.

is 177. They assume that 8 512-bit blocks can be processed at the same time. However, these blocks should be of different messages because of the sequential nature of HMAC. Furthermore, although they implemented only one algorithm, the area consumption is high compared to other work that implemented the same number of algorithms. Lu and Lockwood [50] implemented HMAC-MD5 and HMAC-SHA-1 separately and included results for each one alone. They did not address the HMAC area or throughput. They only showed area of MD5 and SHA-1 cores and throughput of processing one 512-bit block. We did not include these works in Table 6.8 because we only included work that integrated more than one hash function. Wang et al. [73] estimated the HMAC throughput to be 27.7-34.7 Mbps for SHA-1 and 34.6-43.2 Mbps for MD5. For our results, we selected the general HMAC implementation with key reuse (D2), although most of the other work used the fixed key size specifications. Our throughput shown is for the best case, which equals the throughput of processing one message block.

The estimated power consumption of our HMAC-hash unit is 857 mW (as shown in § 6.1.4). All other works did not include any criteria that helps finding an estimated value for power consumption. The only exception are Wang et al. [73] who indicated that the power of their ASIC core is 14.9 mW. The low power consumption of their core can be justified by being an ASIC as compared to our FPGA one. Furthermore, as power consumption increases with area, our design is expected to consume more power. We consume more area than theirs since we integrate six standard algorithms whereas they integrated only two. In addition to that, the XPower tool we used to estimate power consumption gives the worst case estimate. Wang et al. [73] did not indicate how they got the reported power consumption.

6.3 Concluding Remarks

In this chapter, we analyzed the performance of the proposed unit using four factors: area, delay, throughput, and power consumption. We analyzed the performance of four different

HMAC implementations with respect to key size and reuse. From the analysis, we concluded that it is better to implement the HMAC algorithm for fixed key size as long as security issues regarding key selection are taken into account. Since the key reuse mechanism is more useful with large keys, it is better not to include this feature in the HMAC implementations for fixed key size. However, we recommend including the key reuse mechanism in any implementation of the HMAC algorithm with general key size.

In addition, we discussed the effect of the design enhancements (pipelining, parallelism, and key reuse) applied to the design of the HMAC-hash unit on its performance. We showed how these enhancements increased the throughput of the designed unit with negligible increase in area.

The obtained results of our work have been compared to results reported in previous work. We first compared our work to those implementing more than one hash function on FPGA. Then, we compared our work to those implementing HMAC with some hash functions. The comparison showed that our proposed unit achieved a comparable or better performance than most of the previous work.

The work discussed in this chapter and in Chapters 4 and 5 is integrated in a journal paper submitted for publication in [94].

Chapter 7

Conclusions

This chapter is a conclusion of the work reported in this dissertation. In § 7.1, a summary of the work of this dissertation and the results obtained is given. In § 7.2, we discuss the contributions of this dissertation. § 7.3 highlights some directions for future work.

7.1 Summary

Authentication is an important security service used by IPSec and other applications. The most popular techniques used for authentication are hash functions, both keyed and unkeyed. They are used for message authentication, data integrity, entity authentication, and digital signature. The standard algorithms used for IPSec authentication are HMAC-MD5, HMAC-SHA-1, and HMAC-RIPEMD-160. In addition, MD5, SHA-1, and RIPEMD-160 may be used alone in other security applications.

We proposed a unified algorithm that implements any one of MD5, SHA-1, and RIPEMD-160. We implemented this algorithm in a unified hash engine. We integrated the unified hash engine into a reconfigurable HMAC-hash unit that implements any one of MD5, SHA-1, and RIPEMD-160, HMAC-MD5, HMAC-SHA-1, and HMAC-RIPEMD-160.

We followed an emerging design methodology, which uses the Handel-C language to design a target system and map it directly onto FPGA platforms. Handel-C is a high level language that adds some constructs to conventional C language for hardware implementations. We used the available constructs in Handel-C to conduct a design space exploration

in order to get the most optimal results.

Most of the previous work either considered hash functions alone or considered HMAC with hash functions. This work is the first that implements six standard hash algorithms on one reconfigurable unit. In addition, this is the first work that implements HMAC-RIPEMD-160 on FPGA. Furthermore, this is the only work that uses the design approach of direct mapping from high level languages to FPGA in designing hash units.

Compared to work that integrated more than one hash function for FPGA, we achieved better throughput than those integrating three hash functions or more, and comparable throughput to those integrating two hash functions. We also achieved better frequency than all other work that integrated more than one hash function.

Compared to work that implemented HMAC with some hash functions, we got comparable results. However, this work is the only work that implements HMAC-RIPEMD-160.

The area utilization of the designed unit is less than 33% of the available logic on the FPGA chip we used. Thus, the designed unit can fit on a single FPGA chip as an SoC. In addition, it can be integrated to other systems on the same chip.

7.2 Contributions

In the following subsections, we summarize the contributions of this dissertation.

7.2.1 Designing a Unified Hash Engine

The hash functions MD5, SHA-1, and RIPEMD-160 have a number of similarities that were identified and used to propose a unified algorithm that can perform any one of MD5, SHA-1, and RIPEMD-160. We used this unified algorithm to design a unified hash engine.

7.2.2 Developing an Integrated HMAC-Hash Unit

HMAC is used extensively with the three hash functions. Therefore, it is important to integrate an HMAC engine to the hash engine to execute them when combined with HMAC. We used the above hash engine as a sub-unit and integrated it to an HMAC sub-unit. The resultant unit is a reconfigurable HMAC-hash unit that can perform hash functions alone or with HMAC.

One of the steps of HMAC is to resize the key. This step is specified in the HMAC RFC. However, some applications that use HMAC specify fixed key sizes that do not need resizing. We implemented both versions and elaborated on the performance of each.

7.2.3 Design for Reconfigurability

The designed HMAC-hash unit is reconfigurable at runtime and can perform one of six standard algorithms: MD5, SHA-1, RIPEMD-160, HMAC-MD5, HMAC-SHA-1, and HMAC-RIPEMD-160. This work is the first in the literature that integrates six standard algorithms in one reconfigurable unit. In addition, it is the first work that implements HMAC-RIPEMD-160 on FPGA.

7.2.4 Design Enhancements

We improved the performance of the proposed unit by applying some design enhancements, which are summarized in the following:

1. **Pipelining:**

   In order to reduce the number of clock cycles required for hashing a message, we applied the pipelining principle. Pipelined design improves the throughput, especially for large message size.

2. **Parallelism:**

   We utilized some parallelism inherent in the three hash algorithms to reduce the time required for hashing a message.

3. **Key reuse:**

   We proposed a key reuse mechanism. When the key to be used for the current message is the same as the previous message, there is no need to receive a new key and resize it. We added this idea and compared its performance to the performance without it.

7.2.5 Using the Handel-C Design Methodology

We used Handel-C language to build the unified architecture. Although Handel-C is a high level language, it has the capability to be mapped directly to FPGA. In addition, it has constructs that improve the performance of the design, such as pipelining and parallelism. Moreover, because it is not device-specific, the design can be synthesized to different FPGA chips, allowing more flexibility and speed upgrading.

The designed integrated HMAC-hash unit was built on an FPGA platform. The benefit of FPGA designs is the flexibility and reconfigurability. So, if new standards are defined, it is easy to reconfigure the FPGA to include these new standards.

This design approach of direct mapping from high level languages to FPGA has not been used before in designing hash units.

7.2.6 Design Space Exploration

Using Handel-C is excellent to compare different implementations using different constructs. We used the available constructs of Handel-C to try different implementations of the designed unit in order to get the most optimal results.

7.2.7 Performance Analysis

We analyzed the performance of four different implementations:

1. General key size with no reuse.

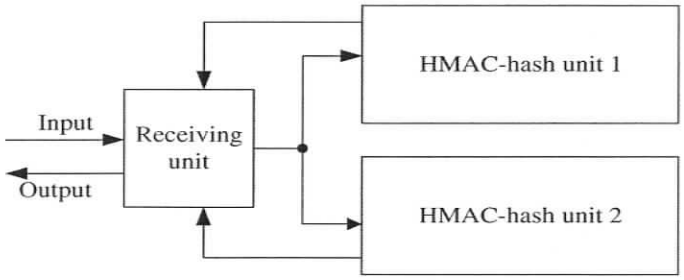

Figure 7.1. *Running replicated HMAC-hash units in parallel to increase throughput.*

2. General key size with reuse.
3. Fixed key size with no reuse.
4. Fixed key size with reuse.

We considered four design factors for analysis: area, delay, throughput, and power consumption.

7.3 Future Work

In the following, we list some suggestions for future work based on the results of this dissertation.

1. In this dissertation, we tried to have a balance between area and throughput. However, if higher throughput is desired, we can replicate the designed unit a number of times, and run these replicated blocks in parallel. This replication increases the overall throughput of the system by using each unit for a different message.

   Figure 7.1 shows an example for duplicating the HMAC-hash unit. The receiving unit is responsible for getting an input message, sending it to one of the two HMAC-hash units, receiving the results from the HMAC-hash unit, and sending the output.

2. We can implement more hash functions. SHA-2 includes three algorithms: SHA-256, SHA-348, and SHA-512. SHA-256 has similarities with SHA-1, whereas SHA-348 and SHA-512 are very similar. We can do one of the following:

- Integrate all these algorithms to our unified hash engine.
- Integrate SHA-256 to the unified engine, and design another hash engine for SHA-348 and SHA-512. The HMAC sub-unit will be interfaced to both engines and selects one of them as required.

3. In this dissertation, we followed a test strategy to test our proposed unit, where we applied standard test vectors that cover all cases of message padding and key resizing. This test strategy is based on a type of hardware testing methodology called test access mechanism (TAM) [95]. For future work, more structured and formal testing methodologies may be applied to ensure the correctness of the functionality and architecture of the designed unit and to have higher test and fault coverage.

   A structured testing methodology could test the designed unit on different levels [96]:

   (a) **Unit level**: where each module or component is tested individually.

   (b) **Integration level**: where some or all components of the designed unit are integrated and tested.

   (c) **System level**: where the designed unit is integrated in a complete system and tested for errors that may not occur in the above two testing levels.

   There are other testing methods and techniques that may be applied to test different aspects of the designed unit. Examples of such test methods and techniques can be found in [95, 97]

4. The main security services provided by AH and ESP are confidentiality and authentication. We can design an IPSec processor that provides these two security services and supports both AH and ESP. This processor consists of two modules. The first one is the HMAC-hash unit designed in this dissertation. The second module is an encryption engine that implements one or more encryption algorithms. Examples of encryption algorithms that are used by IPSec include AES (or Rijndael), DES, and 3DES.

5. The HMAC-hash unit designed in this dissertation is based on hash functions. An-

other popular authentication technique is to use modes of operation that are used with symmetric-key encryption algorithms. We can design a unified engine that implements most of these modes of operations, and then integrate it to the designed HMAC-hash unit to form a cryptographic engine. This cryptographic engine can be used for authentication (using most popular techniques) and for encryption as well.

6. In this dissertation, we integrated six hash algorithms into one reconfigurable, unified hardware unit. Hardware implementations are usually more secure than software implementations. Software implementations encounter a number of vulnerabilities that are eliminated using hardware implementations. Examples of such vulnerabilities include memory access threats, reverse engineering, long-term key storage, and weakness of operating systems security [98].

   However, we need to apply some security test techniques to the proposed unit in order to ensure that the new architecture and implementation are not vulnerable to security threats.

   An example of such security test techniques is to test the key reuse mechanism proposed in this dissertation. To test whether this mechanism affects the security of the designed unit, a certain key could be used for successive HMAC messages and for each message a different hash function is selected. The results of hashing these successive messages are then analyzed to check if any information can be extracted and used to recognize the secret key.

   Another example is to apply some hardware security test techniques to test the security of the hardware architecture of the designed unit. Examples of such techniques include timing analysis, power analysis, fault induction tests [99].

7. There are a number of techniques used for Run-Time Reconfigurability (RTR). Examples include [32]:

   - **Single context:** To use one full configuration at a time.
   - **Multi-context:** Multiple planes of configurations, one of them is active at a time.

- **Partially reconfigurable:** Used in two situations, either when configurations do not occupy the full reconfigurable hardware, or when only some parts of the configurations need modifications.

In this dissertation, we used the first model, where a full configuration is used in a static reconfiguration paradigm. We can try other models, which are especially useful if we integrate more algorithms and the resulted unit cannot fit on a single chip.

8. In this dissertation, we did not optimize the design for power consumption. We can redesign the proposed unit to be power-aware. One of the most important fields that need low power consumption is wireless communication.

9. We can apply the same design methodology we followed in this dissertation to design other cryptographic systems. Examples of cryptographic algorithms that can be targeted are encryption algorithms, such as AES, DES, and RSA, digital signature algorithms, such as DSA, and key agreement protocols, such as Diffie-Hellman.

Bibliography

[1] S. M. Bellovin, "Security problems in the TCP/IP protocol suite," *Computer Communication Review*, vol. 19, no. 2, pp. 32 – 48, Apr. 1998.

[2] G. Schafer, *Security in Fixed and Wireless Networks*. The Atrium, Southern Gate, Chichester, West Sussex, UK: John Wiley & Sons Ltd, Dec. 2003, ch. 11.

[3] Alcatel. (2000, Mar.) Understanding the IPSec protocol suite. Technical Paper. [Online]. Available: http://vpn.shmoo.com/vpn/ipsec_nn.pdf

[4] A. J. Menezes, P. C. van Oorschot, and S. A. Vanstone, *Handbook of Applied Cryptography*. CRC press, 1997.

[5] C. Madson and R. Glenn. (1998, Nov.) RFC 2403 - the use of HMAC-MD5-96 within ESP and AH. [Online]. Available: http://www.faqs.org/rfcs/rfc2403.html

[6] C. Madson and R. Glenn. (1998, Nov.) RFC 2404 - the use of HMAC-SHA-1-96 within ESP and AH. [Online]. Available: http://www.faqs.org/rfcs/rfc2404.html

[7] A. Keromytis and N. Provos. (2000, June) RFC 2857 - the use of HMAC-RIPEMD-160-96 within ESP and AH. [Online]. Available: http://www.faqs.org/rfcs/rfc2857.html

[8] D. Harkins and D. Carrel. (1998, Nov.) RFC 2409 - the Internet key exchange (IKE). [Online]. Available: http://www.faqs.org/rfcs/rfc2409.html

[9] B. Schineier, *Applied Cryptography, Protocols, Algorithms, and Source Code in C*, 2nd ed. John Wiley & Sons, Inc, 1996.

[10] M. Blaze, J. Ioannidis, and A. Keromytis. (2000, Mar.) RFC 2792 - DSA and RSA key and signature encoding for the keynote trust management system. [Online]. Available: http://www.faqs.org/rfcs/rfc2792.html

[11] K. Compton and S. Hauck, "Reconfigurable computing: A survey of systems and software," *ACM Computing Surveys*, vol. 34, no. 2, pp. 171–210, June 2002.

[12] M. K. Jaina, M. Balakrishnan, and A. Kumar, "ASIP design methodologies: Survey and issues," in *Proceedings of the 14th International Conference on VLSI Design (VLSID '01)*, Jan. 2001, pp. 76 – 81.

[13] R. Tessier and W. Burleson, "Reconfigurable computing for digital signal processing:

A survey," *Journal of VLSI Signal Processing*, vol. 28, no. 1-2, pp. 7–27, May-June 2001.

[14] J. Jussel. (2005, Feb.) C to FPGA: An abstract concept for concrete design implementation. RTC Magazine. [Online]. Available: http://www.rtcmagazine.com/home/article.php?id=100304

[15] Altera. (2002, Aug.) FPGAs provide reconfigurable DSP solutions. White Paper, ver 1.0. [Online]. Available: http://www.altera.com/literature/wp/wp_dsp_fpga.pdf

[16] Celoxica Limited. (2003, Dec.) Survey of system design trends. [Online]. Available: http://www.celoxica.com/techlib/files/CEL-W040216Z08-256.pdf

[17] Celoxica Limited. (2002, Aug.) Handel-C language overview. Product Brief. [Online]. Available: http://www.celoxica.com/techlib/files/CEL-W0307171KDD-47.pdf

[18] Celoxica Limited. (2003) Handel-C language reference manual. [Online]. Available: http://www.celoxica.com/techlib/files/CEL-W030811132Q-60.pdf

[19] M. Aubury, I. Page, G. Randall, J. Saul, and R. Watts. (1996, Aug.) Handel-C language reference guide. [Online]. Available: http://www.inf.pucrs.br/~moraes/topicos/hdls/HANDEL-C/HANDELC.PDF

[20] S. Kent and R. Atkinson. (1998, Nov.) RFC 2401 - security architecture for the Internet protocol. [Online]. Available: http://www.faqs.org/rfcs/rfc2401.html

[21] S. Kent and R. Atkinson. (1998, Nov.) RFC 2402 - IP authentication header. [Online]. Available: http://www.faqs.org/rfcs/rfc2402.html

[22] S. Kent and R. Atkinson. (1998, Nov.) RFC 2406 - IP encapsulating security payload (ESP). [Online]. Available: http://www.faqs.org/rfcs/rfc2406.html

[23] D. Maughan, M. Schertler, M. Schneider, and J. Turner. (1998, Nov.) RFC 2408 - Internet security association and key management protocol (ISAKMP). [Online]. Available: http://www.faqs.org/rfcs/rfc2408.html

[24] S. Bakhtiari, R. Safavi-Naini, and J. Pieprzyk, "Cryptographic hash functions: A survey," Department of Computer Science, University of Wollongong, Australia, Tech. Rep. 95-09, July 1995.

[25] NIST. (2002, Mar.) The keyed-hash message authentication code (HMAC). FIPS PUB 198. [Online]. Available: http://csrc.nist.gov/publications/fips/fips198/fips-198a.pdf

[26] H. Krawczyk, M. Bellare, and R. Canetti. (1997, Feb.) RFC 2104 - HMAC: Keyed-hashing for message authentication. [Online]. Available: http://www.faqs.org/rfcs/rfc2104.html

[27] R. Rivest. (1992, Apr.) RFC 1321 - the MD5 message-digest algorithm. [Online]. Available: http://www.faqs.org/rfcs/rfc1321.html

[28] NIST. (2002, Aug.) Secure hash standards. FIPS PUB 180-2. [Online]. Available: http://www.csrc.nist.gov/publications/fips/fips180-2/fips180-2withchangenotice.pdf

[29] D. Eastlake. (2001, Sept.) RFC 3174 - US secure hash algorithm 1 (SHA1). [Online]. Available: http://www.faqs.org/rfcs/rfc3174.html

[30] H. Dobbertin, A. Bosselaers, and B. Preneel. (1996, Apr.) RIPEMD-160, a strengthened version of RIPEMD. [Online]. Available: http://www.esat.kuleuven.ac.be/~cosicart/pdf/AB-9601/AB-9601.pdf

[31] B. Preneel, H. Dobbertin, and A. Bosselaers, "The cryptographic hash function RIPEMD-160," *CryptoBytes*, vol. 3, no. 2, pp. 9 – 14, 1997.

[32] D. Soudris. (2004, Feb.) Survey of FPGA reconfigurable systems: Hardware platforms and software. Democritus University of Thrace. Greece. [Online]. Available: http://www.pldworld.ws/html/technote/patmos.vlsi.ee.upatras.gr/soudris_81DUTH_Contribution.pdf

[33] A. Sutcliffe, *Using Handel-C with DK*, Training Course Manual, Version 1.0.2, Celoxica Limited, Oct. 2004.

[34] E. Sanchez, M. Sipper, J.-O. Haenni, J.-L. Beuchat, A. Stauffer, and A. Perez-Uribe, "Static and dynamic configurable systems," *IEEE Transactions on Computers*, vol. 48, no. 6, pp. 556 – 564, June 1999.

[35] S. Neema, T. Bapty, and J. Scott, "Adaptive computing and runtime reconfiguration," in *2nd Annual Military and Aerospace Applications of Programmable Devices and Technologies Conference (MAPLD)*, Sept. 1999.

[36] J.-C. Lin, C.-T. Chang, and W.-T. Chung, "Design, implementation and performance evaluation of IP-VPN," in *Proceedings of the 17th International Conference on Advanced Information Networking and Applications (AINA 2003)*, Mar. 2003, pp. 206 – 209.

[37] S. J. Leffler, "Fast IPSec: A high-performance IPsec implementation," in *Proceedings of BSDCon 03*, Sept. 2003, pp. 133–140.

[38] S. J. Leffler, "Cryptographic device support for FreeBSD," in *Proceedings of BSDCon 03*, Sept. 2003, pp. 69 – 78.

[39] M. Barton, D. Atkins, J. Lee, S. Narain, D. Ritcherson, K. E. Tepe, and K. D. Wong, "Integration of IP mobility and security for secure wireless communications," in *Proceedings of IEEE International Conference on Communications (ICC 2002)*, vol. 2, Apr.-May 2002, pp. 1045 – 1049.

[40] Y. H. O. Cheung, "Implementation of an FPGA based accelerator for virtual private networks," M. Sc. Thesis, The Chinese University of Hong Kong, July 2002. [Online]. Available: http://www.cse.cuhk.edu.hk/~phwl/mt/public/archives/students/yhcheung.pdf

[41] Y. H. O. Cheung and P. Leong, "Implementation of an FPGA based accelerator for virtual private networks," in *Proceedings of IEEE International Conference on Field-Programmable Technology (FPT), 2002*, Dec. 2002, pp. 34 – 41.

[42] A. Dandalis, V. K. Prasanna, and J. D. P. Rolim, "An adaptive cryptographic engine for IPSec architectures," in *Proceedings of the 2000 IEEE Symposium on Field-Programmable Custom Computing Machines*, Apr. 2000, pp. 132 – 141.

[43] A. Dandalis and V. K. Prasanna, "An adaptive cryptographic engine for Internet protocol security architectures," *ACM Transactions on Design Automation of Electronic Systems*, vol. 4, no. 3, pp. 333 – 353, July 2004.

[44] A. J. Elbirt, "Reconfigurable computing for symmetric-key algorithms," Ph.D. Dissertation, Worcester Polytechnic Institute, Apr. 2002. [Online]. Available: http://faculty.uml.edu/aelbirt/thesis.pdf

[45] P. Chodowiec, K. Gaj, P. Bellows, and B. Schot, "Experimental testing of the gigabit IPSec-compliant implementations of Rijndael and triple DES using SLAAC-1V FPGA accelerator board," in *Proceedings of the Information Security Conference*, Oct. 2001, pp. 220–234.

[46] K.-Y. Lee and J.-C. Kwak, "FPGA implementation of a cryptographic accelerator for IPSec authentications," in *Proceedings of the 2002 International Technical ConferenceOn Circuits/Systems, Computers and Communications (ITC-CSCC 2002)*, July 2002, pp. 948 – 950.

[47] Y. K. Kang, D. W. Kim, T. W. Kwon, and J. R. Choi, "An efficient implementation of hash function processor for IPSec," in *Proceedings of the 2002 IEEE Asia-Pacific Conference on ASIC*, Taipei, Taiwan, Aug. 2002, pp. 93–96.

[48] C.-S. Ha, J. H. Lee, D. S. Leem, M.-S. Park, and B.-Y. Choi, "ASIC design of IPSec hardware accelerator for network security," in *Proceedings of the 2004 IEEE Asia-Pacific Conference on Advanced System Jntegrated Circuits (AP-ASIC 2004)*, Aug. 2004, pp. 168 – 171.

[49] M. McLoone and J. V. McCanny, "A single-chip IPSec cryptographic processor," in *Proceedings of the IEEE Workshop on Signal Processing Systems, SIPS 2002*, Oct. 2002, pp. 133–138.

[50] J. Lu and J. Lockwood, "IPSec implementation on Xilinx Virtex-II Pro FPGA and

its application," in *Proceedings of 19th IEEE International Parallel and Distributed Processing Symposium (IPDPS'05)*, Apr. 2005, p. 158b.

[51] D. Oliva, R. Buchty, and J. N. Heintze, "AES and the cryptonite crypto processor," in *Proceedings of the 2003 International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES03)*, San Jose, California, USA, Oct. 2003, pp. 198 – 209.

[52] E. Nahum, S. O'Malley, H. Orman, and R. Schroeppel, "Towards high performance cryptographic software," in *Proceedings of the Third IEEE Workshop on the Architecture and Implementation of High Performance Communications Subsystems (HPCS)*, Mystic, CT, Aug. 1995, pp. 69 – 72.

[53] J. D. Touch. (1995, June) RFC 1810 - report on MD5 performance. [Online]. Available: http://www.faqs.org/rfcs/rfc1810.html

[54] J. D. Touch, "Performance analysis of MD5," in *Proceedings of the ACM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM '95)*, Aug. 1995, pp. 77–86.

[55] J. M. Arnold, "Mapping the MD5 hash algorithm onto the NAPA architecture," in *Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines, 1998*, Apr. 1998, pp. 267 – 268.

[56] J. Deepakumara, H. M. Heys, and R. Venkatesan, "FPGA implementation of MD5 hash algorithm," in *Proceedings of the Canadian Conference on Electrical and Computer Engineering, 2001*, May 2001, pp. 919 – 924.

[57] K. Jarvinen, M. Tommiska, and J. Skytta, "Hardware implementation analysis of the MD5 hash algorithm," in *Proceedings of the 38th Annual Hawaii International Conference on System Sciences (HICSS '05)*, Jan. 2005, p. 298a.

[58] D. Zibin and Z. Ning, "FPGA implementation of SHA-1 algorithm," in *Proceedings of the 5th International Conference on ASIC, 2003*, vol. 2, Oct. 2003, pp. 1321 – 1324.

[59] P. Kitsos, N. Sklavos, and O. Koufopavlou, "An efficient implementation of the digital signature algorithm," in *Proceedings of the 9th International Conference on Electronics, Circuits and Systems, 2002*, vol. 3, Sept. 2002, pp. 1151 – 1154.

[60] S. Pongyupinpanich, P. Noo-intara, and S. Choomchuay, "SHA-1 with on the fly round-word computation," in *Proceedings of the First Electrical Engineering/Electronics, Computer, Telecommunications, and Information technology Annual Conference (ECTI-CON2004)*, May 2004, pp. 137 – 140.

[61] S. Pongyupinpanich and S. Choomchuay, "An architecture for a SHA-1 applied

for DSA," in *Proceedings of the First Electrical Engineering/Electronics, Computer, Telecommunications, and Information technology Annual Conference (ECTI-CON2004)*, May 2004, pp. 133 – 136.

[62] S. Pongyupinpanich and S. Choomchuay, "An architecture for a SHA-1 applied for DSA," in *Proceedings of the 3rd Asian International Mobile Computing Conference (AMOC 2004)*, May 2004, pp. 8 – 12.

[63] T. Grembowski, R. Lien, K. Gaj, N. Nguyen, P. Bellows, J. Flidr, T. Lehman, and B. Schott, "Comparative analysis of the hardware implementations of hash functions SHA-1 and SHA-512," in *Proceedings of the 5th International Conference on Information Security*, Sept.-Oct. 2002, pp. 75–89.

[64] N. Sklavos, P. Kitsos, K. Papadomanolakis, and O. Koufopavlou, "Random number generator architecture and VLSI implemetation," in *Proceedings of IEEE International Symposium on Circuits and Systems (ISCAS'02)*, vol. 4, May 2002, pp. 854 – 857.

[65] C.-W. Ng, T.-S. Ng, and K.-W. Yip, "A unified architecture of MD5 and RIPEMD-160 hash algorithms," in *Proceedings of the 2004 International Symposium on Circuits and Systems, ISCAS '04*, May 2004, pp. 23–26.

[66] A. Bosselaers, R. Govaerts, and J. Vandewalle, "Fast hashing on the Pentium," in *Advances in Cryptology - CRYPTO 1996*, ser. Lecture Notes in Computer Science. Springer-Verlag, 1996, pp. 298 – 312.

[67] M. Roe, "Performance of block ciphers and hash functions, one year later," in *Proceedings of the 2nd International Workshop for Fast Software Encryption*, Dec. 1994, pp. 359 – 362.

[68] M. Roe, "Performance of block ciphers and hash functions," in *Proceedings of the 1st International Workshop for Fast Software Encryption*, Dec. 1993, pp. 83 – 89.

[69] A. Satoh and T. Inoue, "ASIC-hardware-focused comparison for hash functions MD5, RIPEMD-160, and SHS," in *Proceedings of the International Conference on Information Technology: Coding and Computing (ITCC 2005)*, vol. 1, Apr. 2005, pp. 532 – 537.

[70] S. Dominikus, "A hardware implementation of MD4-family hash algorithms," in *Proceedings of the 9th International Conference on Electronic, Circuits and Systems*, Sept. 2002, pp. 1143–1146.

[71] G. Selimis, N. Sklavos, and O. Koufopavlou, "VLSI implementation of the keyed-hash message authentication code for the wireless application protocol," in *Proceedings of the 10th IEEE International Conference on Electronics, Circuits and Systems, ICECS 2003*, Dec. 2003, pp. 24–27.

[72] H. Michail, A. Kakarountas, A. Milidonis, and C. Goutis, "Efficient implementation of the keyed-hash message authentication code (HMAC) using the SHA-1 hash function," in *Proceedings of the 11th IEEE International Conference on Electronics, Circuits and Systems (ICECS 2004)*, Dec. 2004, pp. 567 – 570.

[73] M.-Y. Wang, C.-P. Su, C.-T. Huang, and C.-W. Wu, "An HMAC processor with integrated SHA-1 and MD5 algorithms," in *Proceedings of the Asia and South Pacific Design Automation Conference, ASP-DAC 2004*, Jan. 2004, pp. 456–458.

[74] M. Oehler and R. Glenn. (1997, Feb.) RFC 2085 - HMAC-MD5 IP authentication with replay prevention. [Online]. Available: http://www.faqs.org/rfcs/rfc2085.html

[75] E. Khan, M. W. El-Kharashi, F. Gebali, and M. Abd-El-Barr, "An FPGA design of a unified hash engine for IPSec authentication," in *the 5th International Workshop on System-on-Chip for Real-Time Applications (IWSOC' 05)*, Banff, Alberta - Canada, July 2005, pp. 450–453.

[76] E. Khan, M. W. El-Kharashi, F. Gebali, and M. Abd-El-Barr, "A reconfigurable hardware unit for the HMAC algorithm," in *the 3rd International Conference on Information & Communication Technology (ICICT 2005)*, Cairo, Egypt, Dec. 2005, accepted for publication.

[77] Celoxica Limited. (2005, July) Software product description for DK version 4.0. [Online]. Available: http://www.celoxica.com/support/articles/521/CEL-ENGSPDDKDK4.0_Software_Product_Description-01001.pdf

[78] Celoxica Limited. (2005, Oct.) Software product description for PDK version 4.0. [Online]. Available: http://www.celoxica.com/support/articles/450/CEL-ENGSPDPDKPDK_4.0_Software_Product_Description-01003.pdf

[79] Xilinx. (2005) Xilinx ISE 7 software manuals and help. [Online]. Available: http://toolbox.xilinx.com/docsan/xilinx7/books/manuals.pdf

[80] Xilinx. Timing Analyzer help. [Online]. Available: http://toolbox.xilinx.com/docsan/xilinx7/help/iseguide/mergedProjects/timingan/timingan.htm

[81] Xilinx. Development system reference guide: XPower. [Online]. Available: http://toolbox.xilinx.com/docsan/xilinx7/de/dev/xpower.pdf

[82] T. Stoecklein and J. Baesig. Handel-C – an effective method for designing FPGAs (and ASICs). [Online]. Available: http://www.celoxica.com/techlib/files/CEL-W0307171HTM-16.pdf

[83] Xilinx. (2005, Mar.) Virtex-II platform FPGAs: Complete data sheet. Product Specification, DS031 (v3.4). [Online]. Available:

http://www.xilinx.com/bvdocs/publications/ds031.pdf

[84] E. Styer. Encryption examples: MD5. Computer Science Department, Eastern Kentucky University. [Online]. Available: http://www.cs.eku.edu/faculty/styer/460/Encrypt/JS-md5.html

[85] E. Styer. Encryption examples: SHA-1. Computer Science Department, Eastern Kentucky University. [Online]. Available: http://www.cs.eku.edu/faculty/styer/460/Encrypt/JS-SHA1.html

[86] P. Cheng and R. Glenn. (1997, Sept.) RFC 2202 - test cases for HMAC-MD5 and HMAC-SHA-1. [Online]. Available: http://www.faqs.org/rfcs/rfc2202.html

[87] J. Kapp. (1998, Feb.) RFC 2286 - test cases for HMAC-RIPEMD160 and HMAC-RIPEMD128. [Online]. Available: http://www.faqs.org/rfcs/rfc2286.html

[88] A. C. J. Kienhuis, "Design space exploration of stream-based dataflow architectures: Methods and tools," Ph.D. Dissertation, Delft University of Technology, The Netherlands, Jan. 1999. [Online]. Available: http://ptolemy.eecs.berkeley.edu/~kienhuis/ftp/thesis.pdf

[89] Xilinx. (2005, Mar.) Virtex-II platform FPGA user guide. UG002 (v2.0). [Online]. Available: http://www.xilinx.com/bvdocs/userguides/ug002.pdf

[90] Celoxica Limited. (2001) Implementing efficient loops in Handel-C. Application Note 71, v1.1. [Online]. Available: http://www.celoxica.com/techlib/files/CEL-W0307171JLS-34.pdf

[91] E. Khan, M. W. El-Kharashi, F. Gebali, and M. Abd-El-Barr, "Design and implementation of an HMAC-hash unit on FPGA using the Handel-C design flow," in *the 2006 IEEE Canadian Conference on Electrical and Computer Engineering (CCECE 2006)*, Ottawa, Canada, May 2006, submitted for publication.

[92] E. Khan, M. W. El-Kharashi, F. Gebali, and M. Abd-El-Barr, "Applying the Handel-C design flow in designing an HMAC-hash unit on FPGAs," *IEE Proceedings Computers & Digital Techniques*, submitted for publication.

[93] E. Khan, M. W. El-Kharashi, F. Gebali, and M. Abd-El-Barr, "Design space exploration of a reconfigurable HMAC-hash unit," *the Elsevier Journal of Systems Architecture (JSA)*, submitted for publication.

[94] E. Khan, M. W. El-Kharashi, F. Gebali, and M. Abd-El-Barr, "Design and performance analysis of a reconfigurable HMAC-hash unit," *ACM Transactions on Architecture and Code Optimization (TACO)*, submitted for publication.

[95] Y. Zorian, E. J. Marinissen, and S. Dey, "Testing embedded-core based system chips," *Computer*, vol. 32, no. 6, pp. 52–60, June 1999.

[96] G. N. Brower, "Software structural testing methods," *Journal of Validation Technology*, vol. 32, no. 6, pp. 37–42, Nov. 1998.

[97] A. Gupta, "Formal hardware verification methods: A survey," *Formal Methods in System Design*, vol. 1, pp. 151–238, 1992.

[98] H. Bar-El. (2002, Oct.) Security implications of hardware vs. software cryptographic modules. White Paper. Discretix Technologies Ltd. [Online]. Available: http://www.hbarel.com/publications/ Security_Implications_of_HW_vs_SW_Cryptographic_Modules.pdf

[99] P. Kocher, R. Lee, G. McGra, A. Raghunathan, and S. Ravi, "Security as a new dimension in embedded system design," in *Proceedings of the 41st Conference on Design Automation (DAC'04)*, June 2004, pp. 753 – 760.

[1] All online references have been visited on November 23, 2005.