DESIGN AND FPGA IMPLEMENTATION OF HASH PROCESSOR

A THESIS SUBMITTED TO
THE GRADUATE SCHOOL OF NATURAL AND APPLIED SCIENCES
OF
MIDDLE EAST TECHNICAL UNIVERSITY

BY

TUĞBA ŞİLTU ÇELEBİ

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR
THE DEGREE OF MASTER OF SCIENCE
IN
ELECTRICAL AND ELECTRONICS ENGINEERING

DECEMBER 2007

DESIGN AND FPGA IMPLEMENTATION OF HASH PROCESSOR

Submitted by TUĞBA ŞİLTU ÇELEBİ in partial fulfillment of the requirements for the degree for the degree of **Master of Science in Electrical and Electronics Engineering, Middle East Technical University** by,

Prof. Dr. Canan Özgen ________________
Dean, Graduate School of **Natural and Applied Sciences**

Prof. Dr. İsmet Erkmen ________________
Head of Department, **Electrical and Electronics Engineering Dept.**

Prof. Dr. Murat AŞKAR
Supervisor, **Electrical and Electronics Engineering Dept.** ________________

Examining Committee Members:

Prof. Dr. Rüyal ERGÜL
Electrical and Electronics Engineering Dept., METU ________________

Prof. Dr. Murat AŞKAR
Electrical and Electronics Engineering Dept., METU ________________

Prof. Dr. Hasan GÜRAN
Electrical and Electronics Engineering Dept., METU ________________

Assoc. Prof. Dr Melek YÜCEL
Electrical and Electronics Engineering Dept., METU ________________

Dr. Murat Hamdi YILDIRIM ________________
(BILKENT, CTIS)

Date ________________

I hereby declare that all information in this document has been obtained and presented in accordance with academic rules and ethical conduct. I also declare that, as required by these rules and conduct, I have fully cited and referenced all material and results that are not original to this work.

Name, Last name :Tuğba Şiltu Çelebi

Signature :

ABSTRACT

DESIGN AND FPGA IMPLEMENTATION OF HASH PROCESSOR

ŞİLTU, ÇELEBİ Tuğba
M.Sc., Department of Electrical and Electronics Engineering
Supervisor: Prof. Dr. Murat AŞKAR

December 2007, 119 pages

In this thesis, an FPGA based hash processor is designed and implemented using a hardware description language; VHDL.

Hash functions are among the most important cryptographic primitives and used in the several fields of communication integrity and signature authentication. These functions are used to obtain a fixed-size fingerprint or hash value of an arbitrary long message.

The hash functions SHA-1 and SHA2-256 are examined in order to find the common instructions to implement them using same hardware blocks on the FPGA. As a result of this study, a hash processor supporting SHA-1 and SHA2-256 hashing and having a standard UART serial interface is proposed. The proposed hash processor has 14 instructions. Among these instructions, 6 of them are special instructions developed for SHA-1 and SHA-256 hash functions. The address length of the instructions is six bits. The data length is 32 bits. The proposed instruction set can be extended for other hash algorithms and they can be implemented over the same architecture.

The hardware is described in VHDL and verified on Xilinx FPGAs. The advantages and open issues of implementing hash functions using a processor structure are also discussed.

Keywords: processor, hash function, cryptography, VHDL

ÖZ

GÜVENLİ ÖZETLEME ALGORİTMALARI İŞLEMCİSİ MODELLENMESİ VE FPGA ÜZERİNDE GERÇEKLEŞTİRİLMESİ

ŞİLTU ÇELEBİ, Tuğba

Yüksek Lisans, Elektrik ve Elektronik Mühendisliği Bölümü

Tez Yöneticisi: Prof. Dr. Murat AŞKAR

Aralık 2007, 119 sayfa

Bu tezde, VHDL donanım modelleme dili kullanılarak güvenli özetleme algoritmalarını gerçekleyen FPGA tabanlı bir işlemci tasarlanmış ve gerçekleştirilmiştir.

Güvenli özetleme algoritmaları en temel kriptolojik algoritmalar arasındadır ve iletişim ve imza doğrulama işlemlerinin birçok aşamasında kullanılmaktadır. Bu

fonksiyonlar değişebilir uzunluktaki bir mesajın sabit uzunlukta özetini elde etmek için kullanılmaktadır.

Güvenli özetleme algoritmalarından olan SHA1 ve SHA2–256, her iki algoritmayı da FPGA üzerinde ortak donanım blokları kullanarak gerçekleştirmek için komutlar bulmak amacıyla detaylı ve karşılaştırmalı olarak incelenmiştir. Bu incelemenin sonucunda SHA-1 ve SHA-256 güvenli özetleme algoritmalarını destekleyen ve standart UART iletişim ara yüzüne sahip bir güvenli özetleme algoritması işlemcisi tasarlanmıştır. Güvenli özetleme algoritması işlemcisinin komut seti 14 komuttan oluşmaktadır. Bu komutlardan 6 tanesi SHA-1 ve SHA-256 güvenli özetleme algoritmaları için geliştirilmiş özel komutlardır. Komutların adres boyu 6 bit, veri uzunluğu ise 32 bittir. Tasarlanan komut seti diğer özetleme fonksiyonları için de genişletilebilir ve aynı mimari yapı kullanılarak gerçekleştirilebilir.

Tasarım, VHDL dili kullanılarak modellenmiş ve Xilinx FPGA kullanılarak donanım ortamında doğrulanmıştır. Güvenli özetleme algoritmalarının bir işlemci yapısında gerçekleştirilmesinin avantajları ve dezavantajları vurgulanmıştır.

Anahtar Kelimeler: işlemci, güvenli özetleme algoritması, kriptoloji, VHDL

To My Dear Family

ACKNOWLEDGMENTS

I would like to express my sincere gratitude to my supervisor Prof. Dr. Murat Aşkar for his guidance, valuable ideas and support during this study.

I would like to thank to my colleagues at ASELSAN Inc. for their support, guidance and valuable contribution to this thesis work. I am grateful to ASELSAN Inc. for providing tools and other facilities for the completion of this thesis work.

Finally I want to express my deepest gratitude to my parents Zeynep and Vehbi ŞİLTU, my dear sisters Hilal and Esra ŞİLTU for their priceless support, encouragement and endless love they have given me not only through my thesis work but also through all stages of my life.

Last but not least, I am grateful to my husband Özgür ÇELEBİ not only for his love, great understanding, encouragement and personal sacrifice but also his continuous technical support and guidance through my thesis work.

TABLE OF CONTENTS

PLAGIARISM .... **iii**

ABSTRACT .... **iv**

ÖZ .... **vi**

ACKNOWLEDGMENTS .... **ix**

TABLE OF CONTENTS .... **x**

LIST OF TABLES .... **xii**

LIST OF FIGURES .... **xiii**

LIST OF FIGURES .... **xiii**

LIST OF ABBREVIATIONS .... **xv**

CHAPTER I INTRODUCTION **1**

CHAPTER II HASH FUNCTIONS AND PROCESSORS **6**

2.1 HASH FUNCTIONS 6

2.1.1 DEFINITION AND PROPERTIES OF HASH FUNCTION 6

2.1.2 APPLICATIONS OF HASH FUNCTIONS 9

2.1.3 ATTACKS TO THE HASH FUNCTIONS 16

2.1.4 KNOWN HASH FUNCTIONS 17

2.1.5 HASH COMPUTATION FLOW 20

2.2 DIFFERENT HASH IMPLEMENTATIONS 33

2.2.1 COMMERCIAL HASH FUNCTION IMPLEMENTATIONS 42

CHAPTER III DESIGN OF HASH PROCESSOR **46**

3.1 DESIGN ON FPGA 46

3.1.1 CONFIGURING FPGAS 47

3.2 HASH PROCESSOR IMPLEMENTATION 51

3.2.1 RESOURCES USED IN THE DESIGN 53

3.2.2 HASH PROCESSOR ARCHITECTURE AND INSTRUCTION SET 54
3.2.3 HASH PROCESSOR MODULES 57

CHAPTER IV HARDWARE REALIZATION OF HASH PROCESSOR 74

5.1 HASH PROCESSOR OVER AN FPGA 74
5.2 TEST AND VERIFICATION METHODOLOGY 78
5.3 TEST AND SIMULATION RESULTS 80

CHAPTER V DISCUSSION AND CONCLUSION 85

REFERENCES 88

APPENDICES 91

APPENDIX-A 92

SHA-1 AND SHA-256 CONSTANTS 92

APPENDIX B 95

COMMERCIAL HASH IMPLEMENTATIONS 95

B.1 CAST SHA-1 SECURE HASH FUNCTION CORE 95
B.2 CAST SHA-256 SECURE HASH FUNCTION CORE 96
B.3 HDL DESIGN HOUSE HCR_SHA1 97
B.4 HELION TECHNOLOGY LIMITED SHA-1, SHA-256 AND MD5 HASHING, FAST (HELION) 98
B.5 HELION TECHNOLOGY LIMITED SHA-1, SHA-224, SHA-256 AND MD5 HASHING, TINY WITH HMAC 99
B.6 ALDEC INC ALDEC SHA IP CORE 100
B.7 OCEAN LOGIC PTY. LTD OL_SHA256 SHA-256 PROCESSOR 100
B.8 OCEAN LOGIC PTY. LTD OL_SHA SHA-1 PROCESSOR 101
B.9 SCI-WORX HIGH SPEED SHA-1 HASH ENGINE 102

APPENDIX C 104

STRUCTURE OF CD-ROM DIRECTORY 104

LIST OF TABLES

Table 2-1 Summary of Standard Hash Functions 17
Table 2-2 SHA-1 Summary 21
Table 2-3 SHA-1 Functions 23
Table 2-4 SHA-1 Constants 23
Table 2-5 Initial Hash Value for SHA-1 24
Table 2-6 SHA-256 Summary 27
Table 2-7 Initial Hash Value for SHA-1 30
Table 2-8 Commercial Hash Function Cores 45
Table 3-1 Software Resources Used in the Design 53
Table 3-2 Used Hardware for Verification 53
Table 3-3 Hash Processor Instructions 55
Table 3-4 Input Output Signals of the Control Unit 57
Table 3-5 Opcodes 61
Table 3-6 Input Output Signals of the Program Memory 65
Table 3-7 Input Output Signals of the Message Expansion Block 66
Table 3-8 Input Output Signals of the ROM Block 67
Table 3-9 Input Output Signals of the Register File 67
Table 3-10 Input Output Signals of the ALU 71
Table 3-11 ALU Operation Selection 72
Table 3-12 UART Baud Rate Selection Table 73
Table 5-1 Device Utilization Summary for Hash Processor VHDL Code 78
Table 5-2 SHA-1 Calculation Program 80
Table 5-3 SHA-256 Calculation Program 81
Table A-1 SHA-1Constants 92
Table A-2 SHA-256 Constants 92

LIST OF FIGURES

Figure 2-1 Hashing Operation ... 7
Figure 2-2 Preimage Resistance ... 7
Figure 2-3 Second Preimage Resistance ... 8
Figure 2-4 Collision Resistance ... 8
Figure 2-5 Verifying Data Integrity ... 10
Figure 2-6 Storing the Hash of a Password ... 11
Figure 2-7 Authenticating Users ... 12
Figure 2-8 Application of a Digital Signature ... 14
Figure 2-9 Verification of a Digital Signature ... 15
Figure 2-10 General Hash Computation Flow ... 20
Figure 2-11 Ch Function Architecture ... 22
Figure 2-12 Parity Function Architecture ... 22
Figure 2-13 Maj Function Architecture ... 23
Figure 2-14 Message Padding ... 24
Figure 2-15 SHA-1 Computation Flow ... 26
Figure 2-16 $\sum_0^{256}(x)$ Architecture ... 28
Figure 2-17 $\sum_1^{256}(x)$ Architecture ... 28
Figure 2-18 $\sigma_0^{256}(X)$ Architecture ... 29
Figure 2-19 $\sigma_1^{256}(X)$ Architecture ... 29
Figure 2-20 SHA-256 Computation Flow ... 33
Figure 2-21 General Block Diagram for a Hash Function Implementation ... 35
Figure 2-22 The Block Diagram of Non-Resource Sharing Design [7] ... 36
Figure 2-23 The Block Diagram of Resource Sharing Design [7] ... 37
Figure 2-24 Shift Register Design Approach [8] ... 38

Figure 2-25 Left and Right Datapaths[10] 39
Figure 2-26 Common Architecture for SHA-256, SHA-384 and SHA-512.... 40
Figure 2-27 HashChip Architecture [13] 41
Figure 3-1 FPGA Architecture [28] 46
Figure 3-2 HDL Based FPGA Design Flow 47
Figure 3-3 Schematic Based FPGA Design Flow.. 48
Figure 3-4 Different Levels of Abstraction.. 49
Figure 3-5 VHDL Design Flow Summary [28] 50
Figure 3-6 Block Diagram of a Processor.. 52
Figure 3-7 Hash Processor General Block Diagram..................................... 55
Figure 3-8 Controller State Diagram 64
Figure 3-9 Datapath Architecture 65
Figure 5-1 Xilinx ML402 Evaluation Platform Front Side........................... 76
Figure 5-2 Xilinx ML402 Evaluation Platform Back Side 76
Figure 5-3 Advanced Hash Calculator... 79
Figure 5-4 Hash Processor User Interface 80
Figure 5-5 SHA-256 Calculation for Input “abc” 82
Figure 5-6 SHA-1 Calculation for Input “abc” 83
Figure 5-7 SHA-1 Calculation for Input “tugba”.. 84
Figure 5-8 SHA-1 Output of AHC for Input “tugba” 84
Figure B-1 CAST SHA-1 Secure Hash Function Core Block Diagram........ 95
Figure B-2 CAST SHA-256 Secure Hash Function Core Block Diagram.... 96
Figure B-3 HDL Design House HCR_SHA1 Core Block Diagram............. 97
Figure B-4 Hellion Fast Hashing Core Block Diagram................................ 98
Figure B-5 Hellion Tiny Hashing Core Block Diagram............................... 99
Figure B-6 ALDEC SHA IP Core Block Diagram.................................... 100
Figure B-7 Ocean Logic Pty. Ltd SHA-256 Processor Block Diagram...... 101
Figure B-8 OL_SHA SHA-1 Processor Core Block Diagram 102
Figure B-9 Sci-worx High Speed SHA-1 HASH Engine Block Diagram . 103

LIST OF ABBREVIATIONS

FPGA	Field Programmable Gate Array
HDL	Hardware Description Language
IEEE	Institute of Electrical and Electronics Engineers
UART	Universal Asynchronous Receiver / Transmitter
VHDL	VHSIC Hardware Description Language
SHA	Secure Hash Algorithm
MD	Message Digest
RIPEMD	RACE Integrity Primitives Evaluation Message Digest
NIST	National Institute of Standards and Technology
SHS	Secure Hash Standard
FIPS	Federal Information Processing Standards
ASIC	Application Specific Integrated Circuit
DSA	Digital Signature Algorithm
RTL	Register Transfer Level
UART	Universal Asynchronous Receiver / Transmitter
ROM	Read Only Memory
RAM	Random Access Memory
BRAM	Block Random Access Memory
CLB	Configurable Logic Block
LUT	Look Up Table

CHAPTER I

INTRODUCTION

In this thesis, hash functions SHA-1 and SHA-256 are implemented on FPGA in a processor structure. The design is described and captured using a hardware description language, namely VHDL.

Due to the rapid developments in the wireless communications area and personal communications systems, providing information security has become a more and more important subject. This security concept becomes a more complicated subject when next-generation system requirements and real-time computation speed are considered. In order to solve these security problems, lots of research and development activities are carried out and cryptography has been a very important part of any communication system in the recent years. Cryptographic algorithms fulfill specific information security requirements such as data integrity, confidentiality and data origin authentication [1].

Hash functions are among the most important cryptographic algorithms and used in the several fields of communication integrity and signature authentication. These functions are sort of operations that take an arbitrary length of input and produce a condensed representation of that input. This condensed representation of an arbitrary long input is usually referred as message digest or hash value. The size of the message digest is fixed depending on the particular hash function being used. The security of a hash function is directly related to this message digest length. Hash functions have some specific properties that make

them secure; these properties are pre-image resistance, second pre-image resistance and collision resistance as indicated in the documents of FIPS[1, 2, 3]. Pre-image resistance means that for all predefined hash values it is computationally very hard to find an input having that particular hash value. Second pre-image resistance means that given an input, it is computationally very hard to find another input such that both inputs have the same hash value. Collision resistance means that it is computationally very difficult to find two inputs having the same hash value.

Hash functions are mostly used to provide password authentication in different applications, generating digital signature with DSA (Digital Signature Algorithm) and for verifying data integrity [1]. In order to protect passwords from attacks, hash values of the passwords are stored in the password database rather than clear text. When a user logs into the system, the hash of the password entered by the user is calculated and compared with the one stored in the database. If two hash values match, the user is authenticated; otherwise the user is not granted. In order to generate digital signatures and sign the document with that signature, the hash value of the document is calculated. Then, this calculated hash value is encrypted with a private key/public key using an encryption algorithm. This digital signature is appended to the document and the document is sent with that signature. At the receiving end only the user having the public key/private key related to the person sending the document can decrypt the digital signature and reach to the original hash value. The receiving person then calculates the hash value of the received document. If the two hashes match then both the origin of the document is authenticated and the content of the document is verified [4]. In order to verify data integrity, the hash values of the documents are calculated and kept in a location. Then at a later time, hash value of the document is recomputed. If the hash values do not match one conclude that the file is corrupted [5]. The same technique is used for timestamping the documents.

There are lots of hash functions developed up to now and MD5 (128 bit), SHA-1, SHA-256, SHA-384 and SHA-512 are the most popular of them. The

oldest of these hash functions is the MD5 hash function. This function is developed in 1991 and has an output size of 128 bits [6]. Researches on developing more secure hash functions continued and in 1993 a more secure hash function SHA-1 which provides an output size of 160 bits is developed [2]. In 2002, in order to catch security levels offered by other cryptographic algorithms, NIST developed the three new hash functions: SHA-256, SHA-384 and SHA-512. These hash functions are standardized with SHA-1 as SHS (Secure Hash Standard) [3]. A 224-bit hash function SHA-224, based on SHA-256, has been added to SHS in 2004 [3].

Hash calculations are mainly composed of three sections. In the first part the incoming message is padded and fixed sized message blocks are prepared according to the particular hash function being applied. After these padding operations, the message schedule is prepared. In this state, message block is further divided into sub blocks to be used in each round of the hash calculation process. In the hash calculation process message digest is computed after some specific number of iterations related to the algorithm by using [3]:

(i) Algorithm specific constants

(ii) Message words prepared by the message scheduler

(iii) The chaining variables

Hash functions can be implemented in hardware or software. However, as security and throughput requirements of the systems increase, it is found that software implementations can not provide desired security and throughput values. As a result, it is preferred to implement the hash functions in hardware. There are several hash function implementations in the literature and commercially available in the market. These implementations differ from each other according to the properties such as area, speed and throughput. Kyu *et al.* implemented SHA-1, HAS-160 and MD5 algorithms in a single chip and proposed two architectures one resource sharing and the second non-resource sharing [7]. McLoone *et al.* implemented SHA-512 and SHA-384 on a single chip [8]. The proposed design achieves a throughput of 479 Mbps using a shift register design approach in the

message scheduling part and look up tables for the constants required by the algorithms. Grembowski *et al.* implemented SHA-1 and SHA-512 hash functions separately and compared the implementation results [9]. Sklavos *et al.* implemented SHA-1 and RIPEMD-160 hash functions in the same hardware module [10]. The advantage of the proposed implementation is that it exhibits high throughput due to the pipeline technique used in the design. In an another study, Sklavos *et al.* determined a common architecture for SHA-256, SHA-384 and SHA-512 hash functions and implemented these functions separately [11]. The implementation results of the three functions are compared in the provided security level and in the performance by using hardware terms. Michail *et al*. implemented SHA-1 hash function is in such a way that the throughput of the design is increased by %53 and the power dissipation is kept low [12]. In a recent work on hash function implementations, T.S. Ganesh et al. unify the hash functions MD5, SHA-1 and RIPEMD160 [13]. The design is proposed to exhibit better throughput when compared to the existing hash function implementations.

In this study, hash functions SHA-1 and SHA-256 are implemented in a processor structure. Hash functions SHA-1 and SHA-256 are chosen considering the architectural similarities such as, word size and block size and at the same time some computational differences that make the design not straightforward. Analyzing the hash functions an instruction set is developed. The instruction set consists of 14 instructions. Among these instructions six of them are special instructions developed for SHA-1 and SHA-256 hash functions. The other instructions are general purpose instructions. The address length of the instructions is six bits. The data length is 32 bits. The proposed instruction set can be extended for other hash algorithms and they can be implemented using the same architecture.

The processor has the blocks of general purpose processor; additionally it has two more blocks for preparing message schedule and holding the constants required by the algorithm. The design has a UART module for communication with the external environment. This serial interface is used for filling the program

memory and receiving the incoming message blocks. The processor is fully designed and captured using the hardware description language VHDL. Design is implemented on Xilinx FPGA. For the verification of the design, the test vectors announced by NIST [2] are used. For random inputs, "Advanced Hash Calculator (AHC)" software is used [14] for verification.

The organization of this thesis is as follows. In Chapter 2, background in formation on hash functions is given. Their properties are explained in detail. Hash functions developed up to now are listed and a brief description is given about their history. Types of attack to the hash functions are explained. The computation flow of the hash functions SHA-1 and SHA-256 are described in details. Finally different hash function implementations available in the market and existing in the literature are presented.

Chapter 3 covers full design description of the hash function processor. The design specifications and hardware and software resources used are given. Blocks of the hash function processor are explained in detail.

In Chapter 4, the designed hash function processor is verified on both software and hardware. Simulation results are given in this chapter. The synthesis of the VHDL descriptions of the hash processor, implementation into FPGA and hardware based tests are given at the end of this chapter.

Results of the study are presented in Chapter 5. The followed design steps and methods are discussed and further suggestions are made for the future studies.

CHAPTER II

HASH FUNCTIONS AND PROCESSORS

2.1 HASH FUNCTIONS

2.1.1 DEFINITION AND PROPERTIES OF HASH FUNCTION

A hash function is a sort of operation that takes an input and produces a fixed-size string which is called the hash value. The input string can be of any length depending on the algorithm used. The produced output is a condensed representation of the input message or document and usually called as a message digest, a digital fingerprint or a checksum. The size of the message digest is fixed depending on the particular algorithm being used. This means that for a particular algorithm, all input streams yield an output of same length. Furthermore a very small change in the input results with a completely different hash value. This is known as the avalanche effect [1]. The hashing operation is illustrated below in Figure 2-1.

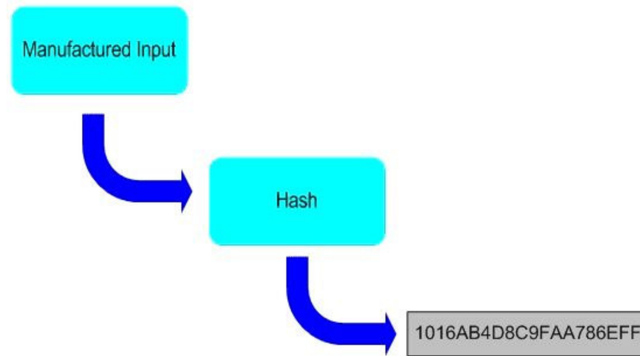

Figure 2-1 Hashing Operation

The security of a hash function is directly related to the message digest length. Pre-image resistance, second pre-image resistance and collision resistance are very important characteristics of any hash function [1].

1. Pre-image resistance (one-wayness): For all specified hash values it is computationally very hard to find an input message having that particular hash value. This property is illustrated in Figure 2-2.

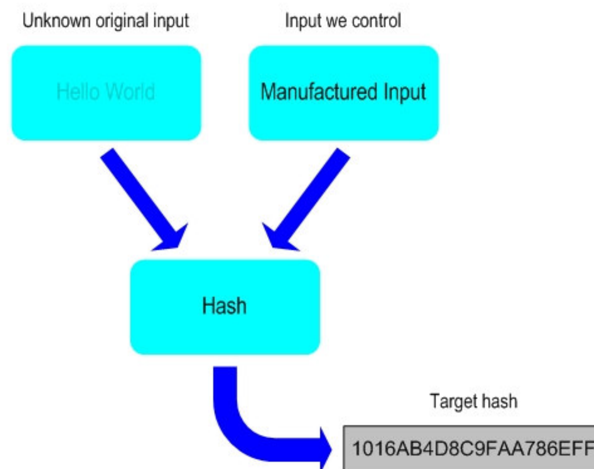

Figure 2-2 Preimage Resistance

2. Second pre-image resistance: Given an input message m_1, it is computationally very hard to find another input message m_2 such that $hash(m_1) = hash(m_2)$. This property is illustrated in Figure 2-3.

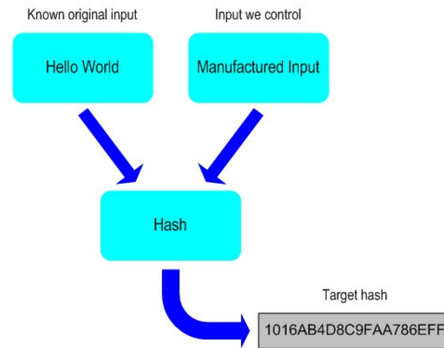

Figure 2-3 Second Preimage Resistance

3. Collision resistance: It is computationally very hard to find any two different inputs that have the same hash value. This property is illustrated in Figure 2-4.

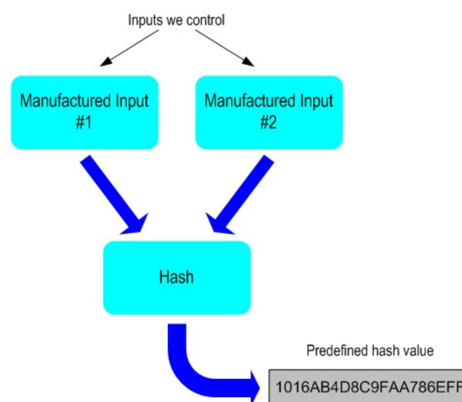

Figure 2-4 Collision Resistance

Hash functions can be classified as keyed and unkeyed hash functions. The keyed hash functions take a secret key as an additional input parameter. In this case, the above defined characteristics of hash functions are satisfied for any value of the secret key. Keyed hash functions are also named as Message Authentication Codes or MACs[1]. In this study, we only deal with unkeyed hash functions.

2.1.2 APPLICATIONS OF HASH FUNCTIONS

The most common use fields of hash functions are verifying data integrity, providing password authentication and generating digital signatures with DSA in applications such as electronic mail, electronic funds transfer, software distribution and data storage which require data integrity assurance and data origin authentication.

Data integrity is a very important part of a secure system. Any changes made to the files can be detected by generating the message digests of the files using a hash function. These digests are saved and in the future the digest is recomputed on the file, if the new digest is different from the original digest, this means that the original file is corrupted some way. This can be very important when protecting critical system binaries and sensitive databases [5]. As an addition during file transmission through the networks such as the internet, files can be corrupted. In order to verify that the received file is identical to the original file, the message digest of the received file is calculated. Then this calculated message digest is compared with the original one published by the WEB site or FTP site. Since it is computationally very hard to find two inputs that have the same hash value (collision resistance property of a hash function), if the calculated digest is different from the original, one can be sure that the received file differs from the transmitted file. Verifying data integrity by means of a hash function is illustrated below in Figure 2-5.

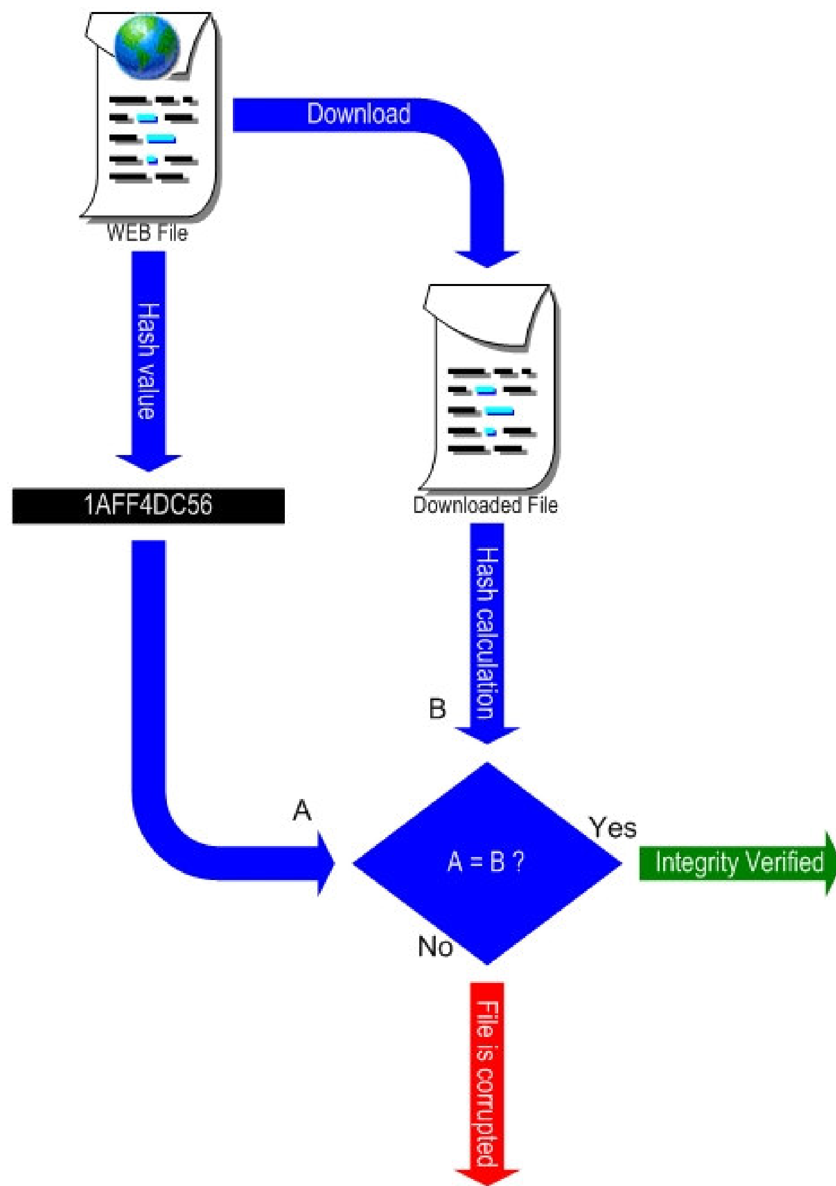

Figure 2-5 Verifying Data Integrity

Password authentication is another field that hash functions are used. For computer systems, it is insecure to store passwords in clear-text. Someone may reach all of the passwords and entire user password database can be compromised. Because of these reasons, a more secure way is to store the hashes of the

passwords rather than clear text passwords. Storing the hashes of passwords is shown below in Figure 2-6.

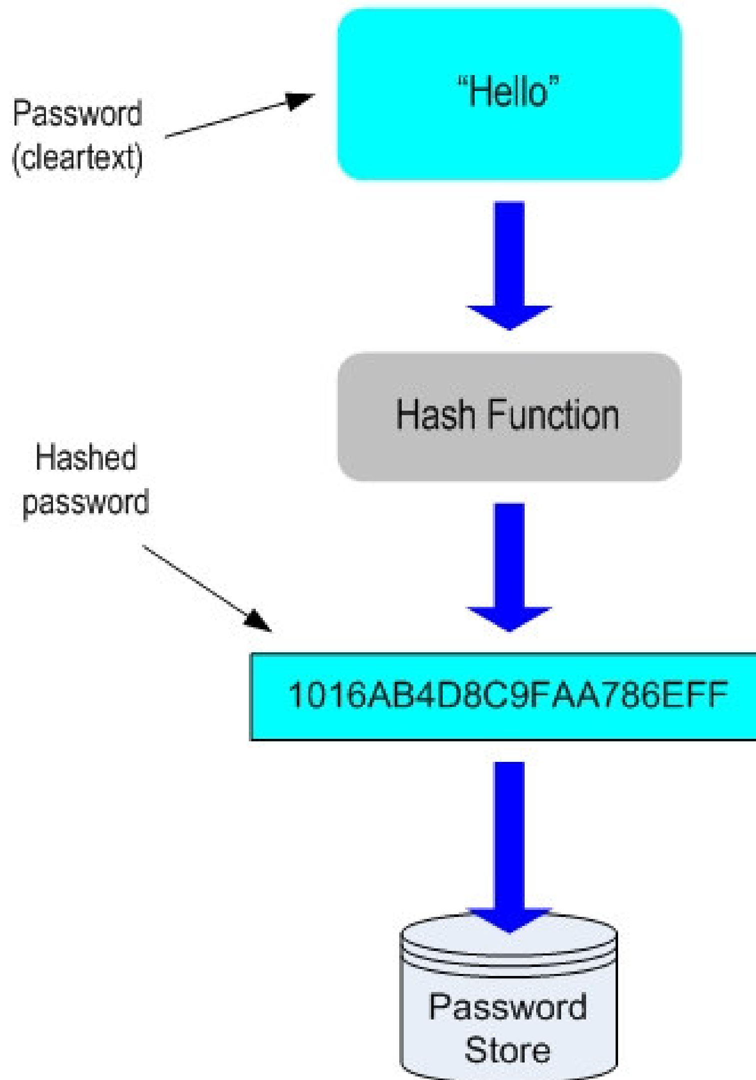

Figure 2-6 Storing the Hash of a Password

When a user logs in, the hash value of the submitted password is calculated and compared with the one stored in the password database. If the calculated hash

value is identical to the one stored in the database, the user is authenticated, and otherwise the user is not granted. This scenario is illustrated below in Figure 2-7.

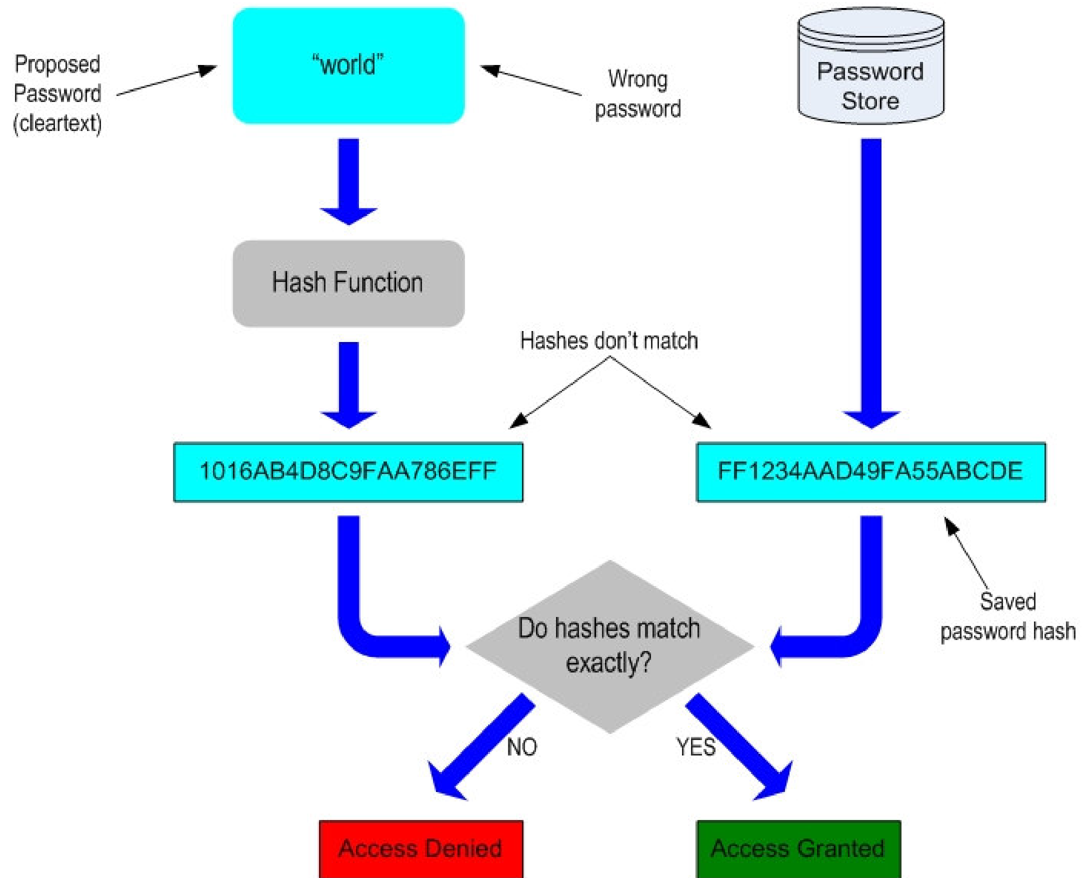

Figure 2-7 Authenticating Users

By this way, even if the password database is compromised, user privacy is still protected since it is computationally very difficult to obtain the original passwords from the hash values.

One of the most popular applications of hash functions is digital signatures. A digital signature is a type of asymmetric cryptography used to simulate the security properties of a signature in digital, rather than in written form

Digital signatures are used to provide authentication of the associated input, usually called a message. Messages can be anything from electronic mail to someone or even a message sent in a more complicated cryptographic protocol. A digital signature scheme consists of three algorithms:

- A key generation algorithm G that randomly produces a "key pair" (PK, SK) for the signer. PK is the verifying key which is to be public and SK is the signing key, to be kept private.
- A signing algorithm S that, on input of a message m and a signing key SK, produces a signature.
- A signature verifying algorithm V that on input a message m, a verifying key PK, and a signature, either accepts or rejects.

Two main properties are required. First, signatures computed properly should always verify. That is, V should accept $(m, PK, S(m, SK))$ where SK is the secret key related to PK, for any message m. Secondly, it should be hard for any adversary, knowing only PK, to create valid signatures [4].

In practice, computing the digital signature of a long message with public key algorithms is very inefficient. To save time, digital signature protocols are often implemented with one-way hash functions [1]. Instead of signing the whole document, hash of the document is signed. In this case, the scenario is as follows:

- The hash value of the document is calculated.
- The calculated hash value is encrypted with the private key, thereby the document is signed
- The document and the signed hash value are send to the recipient
- The recipient calculates the one way hash value of the document and decrypts the signed hash value by using the public key. If the signed hash value is the same with the calculated hash value, then the signature is valid.

The application and verification of a digital signature are illustrated below in Figure 2-8 and Figure 2-9

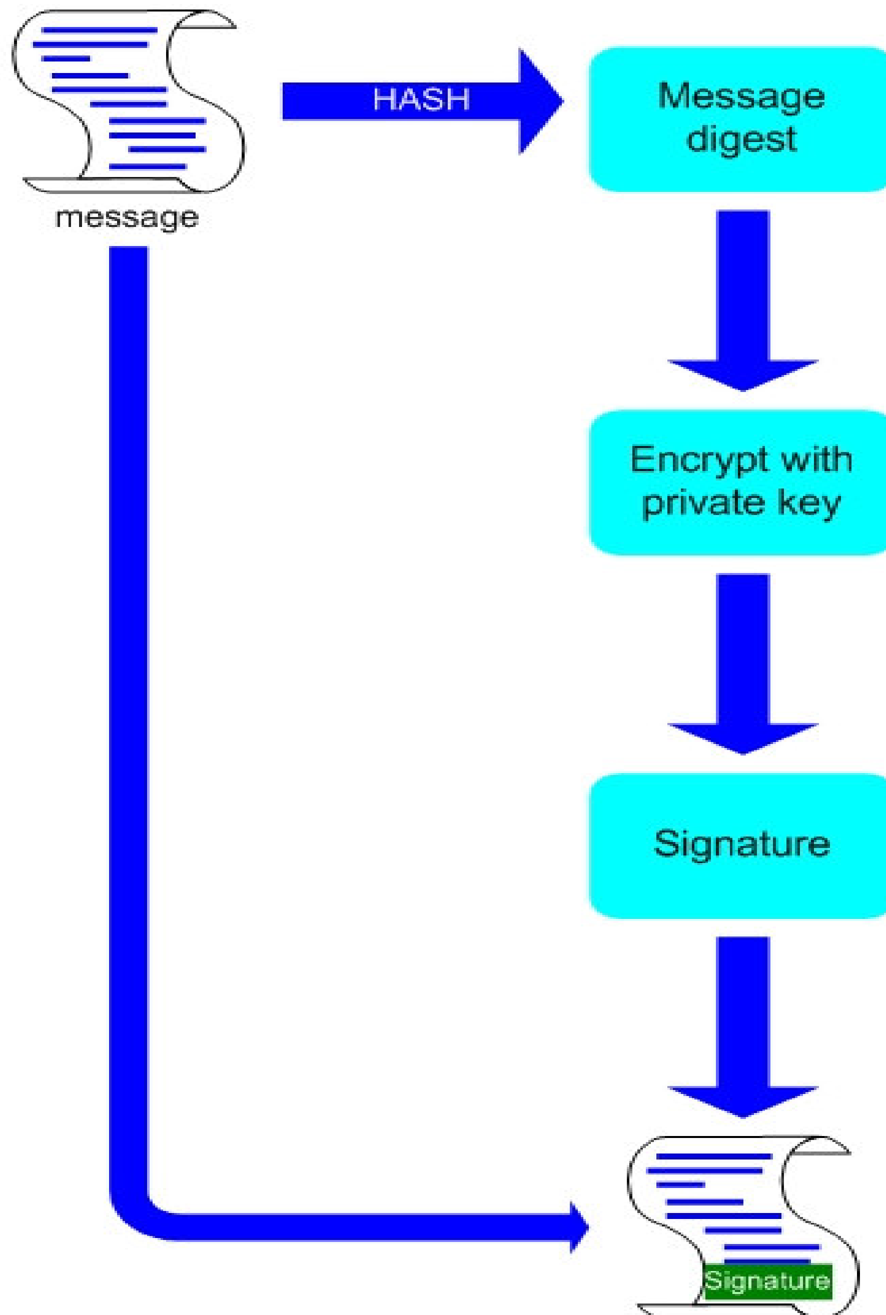

Figure 2-8 Application of a Digital Signature

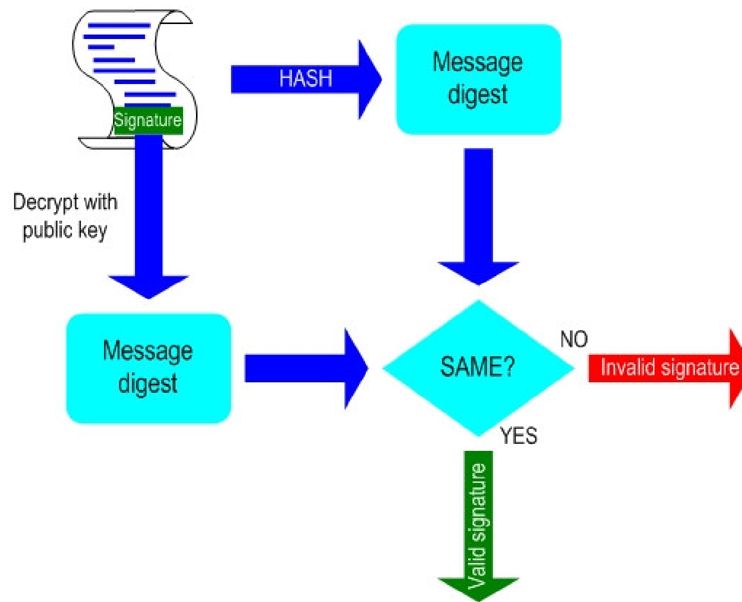

Figure 2-9 Verification of a Digital Signature

If a hash function were not used, the recipient would not be sure that the data integrity is protected. Since hash functions are one way functions, any change in the document will change the signature and the signature would not be validated. As a result, when the signature is validated, the recipient makes sure that the document is not altered. Another benefit of digital signatures is the authentication of the source of the messages. Since private key used in the encryption process belongs to a specific user, a valid signature shows that the message is sent by that user.

One of the earliest proposed applications of digital signatures was to facilitate the verification of nuclear test ban treaties. The United States and Soviet Union (do not exist anymore) permitted each other to put seismometers on the other's soil to monitor nuclear tests. The problem was that each country needed to assure itself that the host nation was not tampering with the data from the monitoring nation's seismometers. Simultaneously, the host nation needed to assure itself that the monitor was sending only the specific information needed for monitoring. Conventional authentication techniques can solve the first problem,

but only digital signatures can solve both problems. The host nation can read but not alter the data from seismometer and the monitoring nation knows that the data has not been tampered with [1].

2.1.3 ATTACKS TO THE HASH FUNCTIONS

There are two brute-force attacks to a hash function [1]. In a brute force, random inputs are tried and the results of the computations are stored until a collision is found [5]. The first attack can be described as follows: Suppose that the hash of a specific message is given, an adversary can try to find another message which has the same hash value. On the other hand, the second attack can be explained as follows: suppose that an adversary tries to find to messages that have the same hash value. This attack is easier than the first one and known as *birthday attack.*

Birthday attack gets its name from the birthday paradox, which is a known statistical problem. The answer to the question, how many people there must be in a room for at least one person sharing your birthday is 183, but surprisingly, the answer to the question how many people there must be in a room for at least two of them will share the same birthday is 23. This means that the probability of two or more people in a group of 23 having the same birthday is greater than ½. Thus, assume that there is a hash function with n-bit output. In order to find a message having a particular hash value, 2^n hash calculations. On the other hand, finding two messages having the same hash value would only require $2^{n/2}$ hash calculations. For instance, a machine which can compute the hash values of one million messages per second would take 600.000 years to find a second message that have a given 64-bit hash value where the same machine can find two messages having the same hash value in about an hour. This means that in order to avoid a birthday attack, someone should choose a hash value twice as long as the actual needed length [1].

2.1.4 KNOWN HASH FUNCTIONS

There is several hash functions developed up to now and among these hash functions MD5, SHA-1, and SHA-256 are most popular. Summary of the standard hash functions is given below in Table 2-1.

Table 2-1 Summary of Standard Hash Functions

Algorithm	Output size	Block size	Word size	Rounds xSteps	Year of the standard
MD4	128	512	32	16x3	1990
MD5	128	512	32	16x4	1991
RIPEMD	128	512	32	16x3 (x2 parallel)	1992
RIPEMD-128	128	512	32	16x4 (x2 parallel)	1996
RIPEMD-160	160	512	32	16x5 (x2 parallel)	1996
SHA-0	160	512	32	80	1993
SHA-1	160	512	32	80	1995
SHA-256	256	512	32	64	2002
SHA-224	224	512	32	64	2004
SHA-384	384	1024	64	80	2002
SHA-512	512	1024	64	80	2002

MD4 proposed by Ron Rivest in 1990 was designed by using 32-bit operations for high speed software implementations on 32-bit processors [15]. MD stands for message digest and the numerals refer to the functions being the fourth design from the same hash function family. However, a collision problem was found and in 1991 MD4 was reformed to MD5 by adding countermeasures such as

increasing the number of compression rounds from three to four [6]. The compression function of MD5 operates on 512 bit blocks and this 512 bit block is further divided into 16 32-bit sub blocks. The word size is 32 bits. There are four 32-bit chaining variables and the output size is 128 bits. One important parameter for compression functions is the number of rounds –the number of sequential updates of the chaining variables. The compression function of MD5 has 64 rounds. MD5 is one of the most popular hash functions for many applications such as IPsec. However it was pointed out that, collisions can be generated using the compression function of MD5 and its 128-bit hash value is not long enough to stop birthday attacks. It was estimated that two messages that have the same hash value could be found within 24 days by developing a dedicated hardware with a cost of 10 million dollars. Considering the processing power of computers is improving 10-fold every 5 years, MD5 is no longer secure against the birthday attack, and it is not recommended for future use.

RIPEMD is a 128 bit hash function developed by the RIPE (RACE Integrity Primitives Evaluation) project in 1992 to address the attack on MD4 [16]. However collisions for the first two and the last two out of three rounds were found. In addition, a 128-bit hash value is no longer secure enough so as described above and thus RIPEMD was improved to the 160-bit hash function RIPEMD-160 in 1996 which has a five round compression function. At the same time, a 128-bit hash function RIPEMD-128 that has a four round compression function was proposed to replace RIPEMD.

NIST (National Institute of Standards and Technology) standardized a 160-bit hash function SHA (Secure Hash Algorithm) for the use with a digital signature algorithm DSS (Digital Signature Standard) in 1993 [2]. Soon after that a way was found to cause collisions in the compression function by analyzing the message expansion function that consisted of only XOR (exclusive OR) operations. In order to modify this SHA was modified to SHA-1 by adding a one-bit rotation to the message expansion function. A 160-bit hash function hash a security level on the order of 80 bits, so SHA-1 is designed to match the security

level of the block cipher Skipjack that uses 80-bit secret key [17]. SHA-1 is modeled taking some cues from MD5, it operates on 512 bit blocks and has five 32 bit chaining variables. The output length is 160 bits. Although the round functions are less varied and simpler than those of MD5, SHA-1 has more rounds –80 instead of 64. SHA-1 uses a more complex procedure for deriving 32-bit sub blocks from the 512 bit message. If one bit of the message is flipped, more than half of the sub blocks get changed, where this number is just four for MD5. In 2001 NIST standardized the new block chipper AES (Advanced Encryption Standard) to replace the DES (Data Encryption Standard) that had been used for more than 20 years [18]. AES supports three key lengths, 128, 192 and 256 bits, whose security levels are higher than SHA-1. In order to match these security levels, NIST developed three new hash functions SHA-256, -384, and -512 whose hash value sizes are 256, 384 and 512 bits, respectively [3]. SHA-256 and SHA-512 have similar designs, with SHA-256 operating on 32-bit words and SHA-512 operating on 64-bit words. Both designs bear strong resemblance to SHA-1 although they are much closer to each other than to their common predecessor. SHA-384 is a trivial modification of SHA-512 which consists of trimming the output to 384-bits and changing the initial value of the chaining variable. These hash functions are standardized with SHA-1 as SHS (Secure Hash Standard) and a 224-bit hash function , SHA-224, based on SHA-256, was added to SHS in 2004. SHA-224 is a truncated version of SHA-256 with a different initial value. The most important difference between the three new functions and SHA-1 is the procedure for deriving 32-bit sub blocks from one block of message. Recently collisions for MD4, MD5, RIPEMD and SHA have been reported and a possibility for breaking SHA-1 has been suggested. Therefore, the migration to more secure hash functions should be accelerated.

In this study, SHA-1 and SHA-256 hash functions are chosen to be implemented as a starting point. The reason for such a selection is that SHA-1 is one of the most commonly used hash functions and SHA-256 is developed after SHA-1 and offers increased security levels. As described above, both of these

functions operate on 512-bit message blocks and word sizes are the same –32 bits. Although they are similar in general, number of chaining variables, the output size, generation of 32-bit sub blocks from 512-bit message blocks and number of rounds differ from each other.

2.1.5 HASH COMPUTATION FLOW

Every hash computation process consists of two stages [2, 3]. The first stage is the preprocessing stage. In this stage the message is padded, parsed into n blocks and the chaining variables are initialized. In the second stage, hash calculation is done. In the hash calculation stage, constants, functions and word operations specific to the hash function are used. Hash calculation generates a message schedule from the padded message and uses that schedule, along with functions, constants and word operations to iteratively generate a series of hash values. The final hash value generated by the hash computation is used to generate the message digest. This scenario is illustrated below in Figure 2-10.

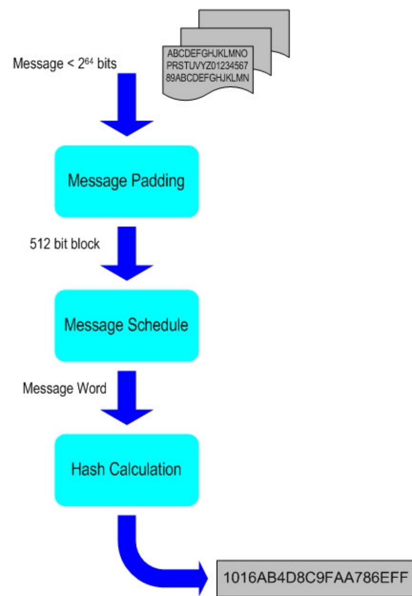

Figure 2-10 General Hash Computation Flow

2.1.5.1 SHA1

SHA1 is one of the most popular hash functions. The message block size for SHA-1 is 512 bits and message digest size is 160 bits. Calculation of message digest for one block message is completed in 80 rounds. The general properties of SHA-1 are summarized in Table 2-2.

Table 2-2 SHA-1 Summary

SHA1	
Message Size	$<2^{64}$
Block Size	512 bits
Word Size	32 bits
Trans.Rounds	80
Message. Digest	160 bits
Security	80 bits
# of chaining variables	5

SHA-1 calculation is completed in 80 rounds and 5 hash variables each of 32 bits are used. The word size of all the calculations is 32 bits. The padded message is processed by 512 bit blocks. This 512 bit block is composed of 16 message words. These 16 message words are expanded by means of functions and in each of the total 80 rounds a new message word is used.

2.1.5.1.1 SHA-1 FUNCTIONS

SHA-1 uses three different logical functions. These functions operate on 32 bit words and each has three parameters. These functions are:

1) $Ch(x, y, z) = (x \wedge y) \oplus (\neg x \wedge z)$

   This function is used in first 20 rounds of SHA-1 calculations. The architecture of this function is illustrated in Figure 2-11.

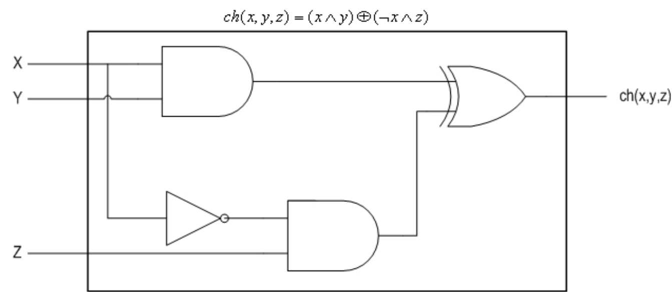

Figure 2-11 Ch Function Architecture

2) $Parity(x, y, z) = x \oplus y \oplus z$

This function is used in second and last 20 rounds of SHA-1 calculations. The architecture of this function is illustrated in Figure 2-12.

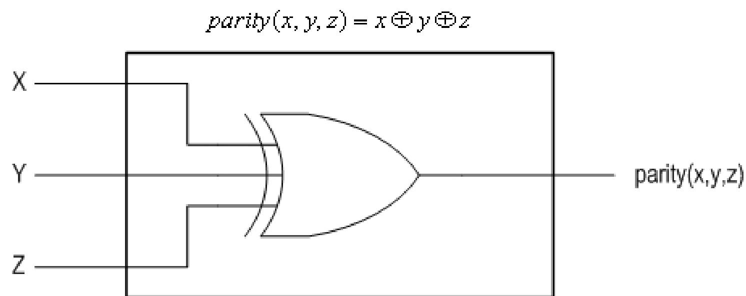

Figure 2-12 Parity Function Architecture

3) $Maj(x, y, z) = (x \wedge y) \oplus (x \wedge z) \oplus (y \wedge z)$

This function is used in third 20 rounds of SHA-1 calculations. The architecture of this function is illustrated in Figure 2-13.

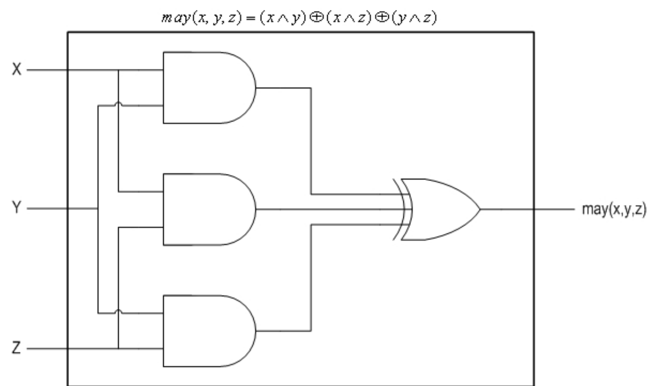

Figure 2-13 Maj Function Architecture

These functions are listed below in Table 2-3 according to the SHA-1 round number.

Table 2-3 SHA-1 Functions

SHA1 Functions	Round number (t)
ft(b, c, d) = Ch(b,c,d)	$0 \leq t \leq 19$
ft(b, c, d) = Parity(b,c,d)	$20 \leq t \leq 39$
ft(b, c, d) = Maj(b,c,d)	$40 \leq t \leq 59$
ft(b, c, d) = Parity(b,c,d)	$60 \leq t \leq 79$

2.1.5.1.2 SHA-1 CONSTANTS

There are four constants which are used in SHA-1 computations. These are given in Table 2-4.

Table 2-4 SHA-1 Constants

SHA1 Constants	Round number (t)
5A827999	$0 \leq t \leq 19$
6ED9EBA1	$20 \leq t \leq 39$
8F1BBCDC	$40 \leq t \leq 59$
CA62C1D6	$60 \leq t \leq 79$

2.1.5.1.3 SHA-1 COMPUTATION FLOW

SHA-1 computation is composed of two stages, preprocessing stage and hash calculation stage. In the preprocessing stage, message is padded, divided into 16 32-bit sub blocks and message schedule is prepared.

- *Message Padding*: Suppose that the length of the message, M, is l bits. Append the bit "1" to the end of the message, followed by *k* zero bits, where k is the smallest, non-negative solution to the equation $l + 1 + k \equiv 448 \bmod 512$. Then append the 64-bit block that is equal to the number *l* expressed using a binary representation. For example, the (8-bit ASCII) message "abc" has length $8x3 = 24$, so the message is padded with a one bit, then $448 - (25 + 1) = 423$ zero bits, and then the message length, to become the 512-bit padded message. This is illustrated below in Figure 2-14.

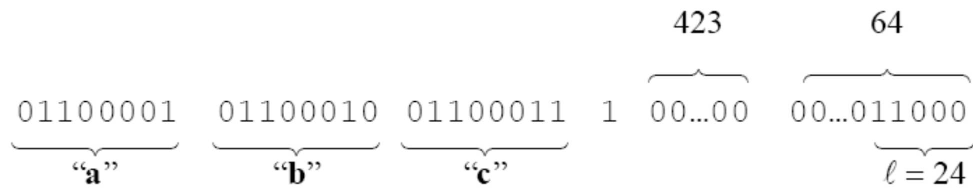

Figure 2-14 Message Padding

- *Setting the initial hash value:* The 160-bit initial hash value $H^{(0)}$ is composed of five 32-bit words which are shown in Table 2-5.

Table 2-5 Initial Hash Value for SHA-1

H0(0)	H1(0)	H2(0	H3(0)	H4(0)
67452301	EFCDAB89	98BADCFE	10325476	C3D2E1F0

- *Hash Calculation*: SHA-1 may be used to hash a message, M, having a length of *l* bits, where $0 \le l \le 2^{64}$. The algorithm uses:

1. A message schedule of 80x32-bit words. The words of the message schedule are labeled W_0 , W_1 , …………W_{80} .
2. Five working variables of 32-bits each. The working variables are labeled as: A, B, C, D, E.
3. A hash value of five 32-bit words. The words of the hash value are labeled as: $H_0^{(i)}$, $H_1^{(i)}$, $H_2^{(i)}$, $H_3^{(i)}$, $H_4^{(i)}$ which will hold the initial hash value $H^{(0)}$, replaced by each intermediate hash value (after each message block is processed) $H^{(i)}$ where i denotes the number of 512 bit block being processed in the message M, and ending with the final hash value, $H^{(N)}$ where N is the number of the last 512 bit block in the message M.
4. A single temporary word, T.
5. Previously defined constants which are labeled K_t where t is the round number.

The calculation is carried out as follows:

The message schedule is prepared, ie. the message word that is going to be used in that round is prepared. This computation is done as described in the following formula:

$$W_t = M_t^i \qquad 0 \le t \le 15$$

$$W_t = ROTL^1(W_{t-3} \oplus W_{t-8} \oplus W_{t-14} \oplus W_{t-16}) \qquad 16 \le t \le 79$$

In the above formula M_t^i denotes the tth 32-bit message word of the ith 512-bit message block in the message M. The 5 working variables A, B, C, D and E that are going to be used in the computation are prepared as follows:

$$A = H_0^{(i-1)}$$

$$B = H_1^{(i-1)}$$

$$C = H_2^{(i-1)}$$

$$D = H_3^{(i-1)}$$

$$E = H_4^{(i-1)}$$

After these initializations, the final values of the working variables for that round are calculated as described below:

$$T = S\,^\wedge 5(A) + f(t;B,C,D) + E + W_t + K_t$$
$$E = D$$
$$D = C$$
$$C = S\,^\wedge 30(B)$$
$$B = A$$
$$A = T$$

As the final step, intermediate hash values are calculated as described below:

$$H_0^{(i)} = A + H_0^{(i-1)}$$
$$H_1^{(i)} = B + H_1^{(i-1)}$$
$$H_2^{(i)} = C + H_2^{(i-1)}$$
$$H_3^{(i)} = D + H_3^{(i-1)}$$
$$H_4^{(i)} = E + H_4^{(i-1)}$$

After 80 rounds the hash value of the incoming 512 bit message block is obtained. Basic SHA-1 computation flow described above is shown below in Figure 2-15:

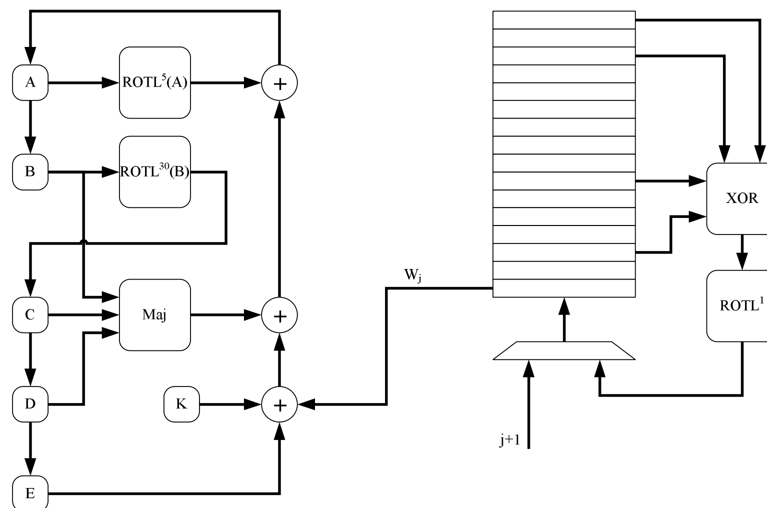

Figure 2-15 SHA-1 Computation Flow

2.1.5.2 SHA-256

SHA-256 is developed after SHA-1 in 2002 by NIST in order to match security levels offered by AES. The message block size for SHA-256 is 512 bits and message digest size is 256 bits. Calculation of message digest is completed in 64 rounds. The general properties of SHA-256 are summarized in Table 2-6:

Table 2-6 SHA-256 Summary

SHA1	
Message Size	$<2^{64}$
Block Size	512 bits
Word Size	32 bits
Trans.Rounds	64
Mes. Digest	256bits
Security	128 bits
# of chaining variables	8

SHA-256 calculation is completed in 64 rounds and 8 hash variables each of 32 bits are used. The word size of all the calculations is 32 bits. The padded message is processed by 512 bit blocks. This 512 bit block is composed of 16 message words. These 16 message words are expanded by means of functions and in each of the total 64 rounds a new message word is used.

2.1.5.2.1 SHA-256 FUNCTIONS

SHA-256 uses six different logical functions. These functions operate on 32 bit words. These functions are:

1) $Ch(x, y, z) = (x \wedge y) \oplus (\neg x \wedge z)$ This function is same as the Ch function used in SHA-1.
2) $Maj(x, y, z) = (x \wedge y) \oplus (x \wedge z) \oplus (y \wedge z)$ This function is same as the Ch function used in SHA-1.

3) $\sum_0^{256}(x) = \mathrm{ROTR}^2(X) \oplus \mathrm{ROTR}^{13}(X) \oplus \mathrm{ROTR}^{22}(X)$ The architecture of this function is shown below in Figure 2-16:

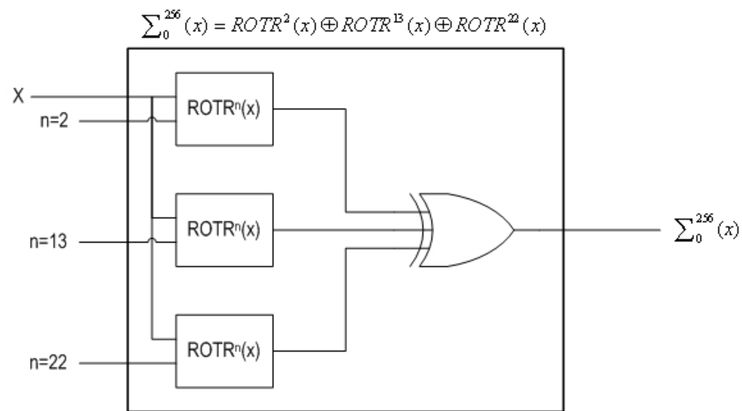

Figure 2-16 $\sum_0^{256}(x)$ **Architecture**

4) $\sum_1^{256}(x) = \mathrm{ROTR}^6(X) \oplus \mathrm{ROTR}^{11}(X) \oplus \mathrm{ROTR}^{25}(X)$ The architecture of this function is shown below in Figure 2-17:

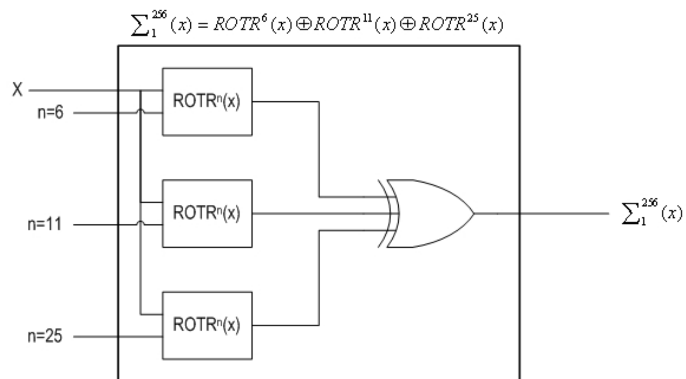

Figure 2-17 $\sum_1^{256}(x)$ **Architecture**

5) $\sigma_0^{256}(X) = \text{ROTR}^7(X) \oplus \text{ROTR}^{18}(X) \oplus SHR^3(X)$ The architecture of this function is shown below in Figure 2-18:

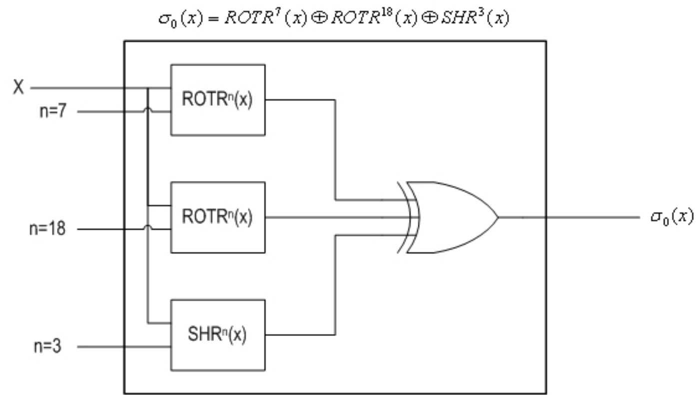

Figure 2-18 $\sigma_0^{256}(X)$ **Architecture**

6) $\sigma_1^{256}(X) = \text{ROTR}^{17}(X) \oplus \text{ROTR}^{19}(X) \oplus SHR^{10}(X)$ The architecture of this function is shown below in Figure 2-19:

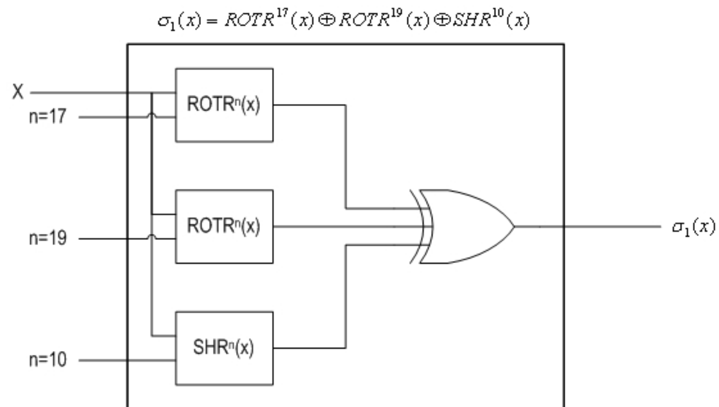

Figure 2-19 $\sigma_1^{256}(X)$ **Architecture**

2.1.5.2.2 SHA-256 CONSTANTS

There are 64 constants which are used in SHA-256 computations. These are given in Appendix A.

2.1.5.2.3 SHA-256 COMPUTATION FLOW

SHA-256 computation is composed of two stages, preprocessing stage and hash calculation stage. In the preprocessing stage, message is padded, divided into 16 32-bit sub blocks and message schedule is prepared.

- *Message Padding*: Message padding operation is done in the same way as in SHA-1.
- *Setting the initial hash value:* The 256-bit initial hash value $H^{(0)}$ is composed of eight 32-bit words which are shown in Table 2-7.

Table 2-7 Initial Hash Value for SHA-1

H0(0)	H1(0)	H2(0)	H3(0)
67452301	BB67AE85	3C6EF372	A54FF53A
H4(0)	**H5(0)**	**H6(0)**	**H7(0)**
510E527F	9B05688C	1F83D9AB	5BE0CD19

- *Hash Calculation*: SHA-256 may be used to hash a message, M, having a length of l bits, where $0 \le l \le 2^{64}$. The algorithm uses:

  1. A message schedule of 64x32-bit words. The words of the message schedule are labeled W_0, W_1, …………W_{64}.
  2. Eight working variables of 32-bits each. The working variables are labeled as: A, B, C, D, E, F, G, H.
  3. A hash value of eight 32-bit words. The words of the hash value are labeled as: $H_0^{(i)}$, $H_1^{(i)}$, $H_2^{(i)}$, $H_3^{(i)}$, $H_4^{(i)}$, $H_5^{(i)}$, $H_6^{(i)}$, $H_7^{(i)}$ which will hold the initial hash value $H^{(0)}$, replaced by each intermediate

hash value (after each message block is processed) $H^{(i)}$ and ending with the final hash value, $H^{(N)}$

4. Two temporary words, T_1 and T_2.
5. Previously defined constants which are labeled K_t, where t is the round number.

The calculation is carried out as follows:

The message schedule is prepared, ie. the message word that is going to be used in that round is prepared. This computation is done as described in the following formula:

$$W_t = \{M_t^i \qquad 0 \le t \le 15$$
$$W_t = \{\sigma_1^{256}(W_{t-2}) + W_{t-7} + \sigma_0^{256}(W_{t-5}) + W_{t-16} \qquad 16 \le t \le 63$$

The eight working variables A, B, C, D, E, F, G and H that are going to be used in the computation are prepared as follows:

$$A = H_0^{(i-1)}$$
$$B = H_1^{(i-1)}$$
$$C = H_2^{(i-1)}$$
$$D = H_3^{(i-1)}$$
$$E = H_4^{(i-1)}$$
$$F = H_5^{(i-1)}$$
$$G = H_6^{(i-1)}$$
$$H = H_7^{(i-1)}$$

After these initializations, the final values of the working variables for that round are calculated as described below:

$$T_1 = H + \sum_1^{256} E + Ch(E, F, G) + K_t + W_t$$
$$T_2 = \sum_0^{256} A + Maj(A, B, C)$$
$$H = G$$
$$G = F$$
$$F = E$$
$$E = D + T_1$$
$$D = C$$
$$C = B$$
$$B = A$$
$$A = T_1 + T_2$$

As the final step, intermediate hash values are calculated as described below.

$$H_0^{(i)} = A + H_0^{(i-1)}$$
$$H_1^{(i)} = B + H_1^{(i-1)}$$
$$H_2^{(i)} = C + H_2^{(i-1)}$$
$$H_3^{(i)} = D + H_3^{(i-1)}$$
$$H_4^{(i)} = E + H_4^{(i-1)}$$
$$H_5^{(i)} = F + H_5^{(i-1)}$$
$$H_6^{(i)} = G + H_6^{(i-1)}$$
$$H_7^{(i)} = H + H_7^{(i-1)}$$

After 64 rounds the hash value of the incoming 512 bit message block is obtained. Basic SHA-256 computation flow described above is shown below in Figure 2-20.

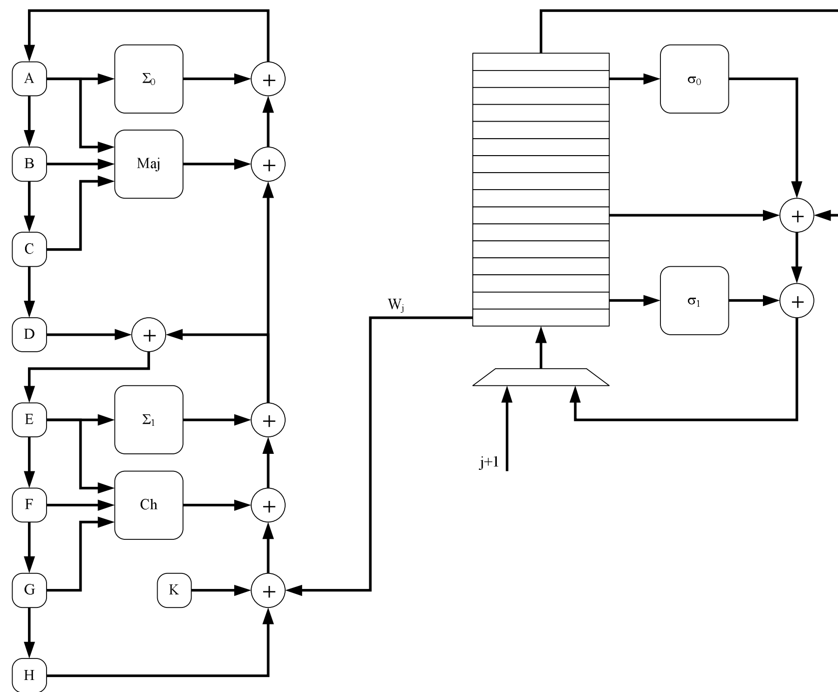

Figure 2-20 SHA-256 Computation Flow

2.2 DIFFERENT HASH IMPLEMENTATIONS

Hash functions can be implemented either in hardware or software. Implementing hash functions completely in software is easier than implementing them in hardware. However, since data rates increase and security protocols become more and more complex, software implementations of hash functions can not satisfy the speed requirements of applications such as embedded systems, network routers and online databases. Furthermore, providing security is another very important issue. System implementation itself should be very secure even if in case of an attack. Software implementations of hash functions can not provide that degree of security since access and modification are easier. When all these aspects are considered, it is seen that it is desirable to implement hash functions in hardware in order to satisfy the speed requirements of the systems and at the same time provide security. Hardware implementations of hash functions are more

secure, since access and modification are harder. Additionally, power consumption is lesser and throughput is higher.

Hardware implementations of hash functions can be divided into two groups: classical implementations and reconfigurable (reprogrammable) implementations. Classical implementations are completely custom designs on Application Specific Integrated Circuits (ASICs) and reconfigurable implementations are on FPGAs. When compared in performance wise, it is found that ASICs exhibit the best performance, FPGAs are close to ASICs and software implementations are the worst of all. On the other hand, when development cost is considered, software development cost is the least, it is a bit higher for FPGAs and development of the ASICs is the most expensive. When considered in terms of flexibility, ASICs are the worst, software implementations are the most flexible ones and FPGAs are close to software implementations since they are reconfigurable structures. According to these judgments, it is obvious that FPGA implementations have the advantages of both hardware and software. Implementation of hash functions on reconfigurable platforms such as FPGAs brings some advantages. These advantages can be listed as follows:

- Ease of algorithm modification: Any modifications can be made easily due to the reconfigurable nature of the FPGAs.
- Architecture efficiency
- Resource efficiency: FGPA implementation of hash functions require less resources in the development phase
- Cost efficiency: FPGA implementations are cost effective since they have shorter design lead time
- High throughput: FPGA implementations work at high speeds, so exhibit high throughput

Hash function implementation on hardware is a very active research area and various implementations exist in the literature. These implementations differ

from each other according to the specifications such as area, speed, throughput, complexity of design and power consumption. Although there are some differences between the implementation of complex arithmetic and logic functions, main hardware blocks in each design are similar. The general block diagram of a hash function implementation is illustrated in Figure 2-21:

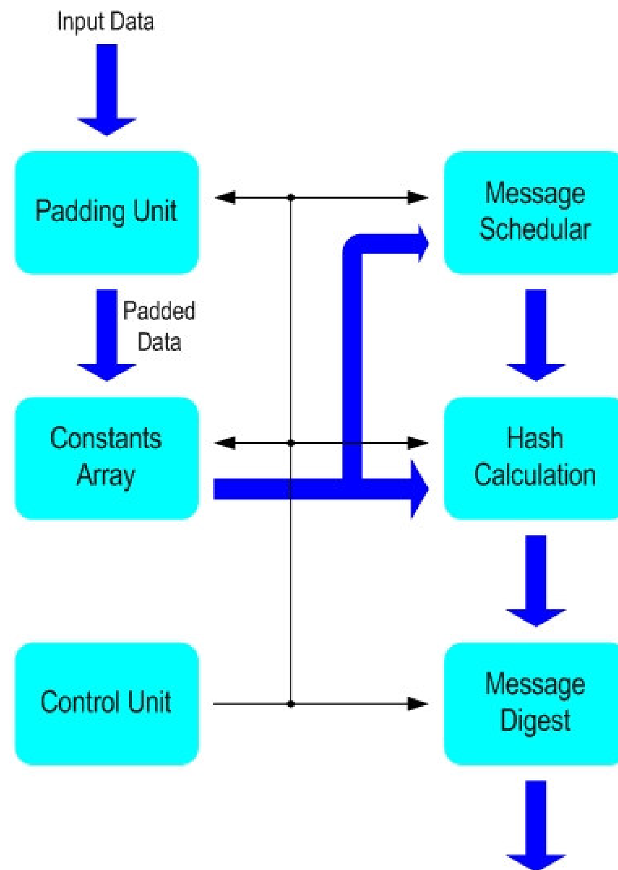

Figure 2-21 General Block Diagram for a Hash Function Implementation

In general the message is input to the hardware as 32-bit message words. The padding unit counts the incoming message words and makes necessary computations described in 2.1.5.1.3 to pad the message and prepare the message blocks. The prepared message block is usually stored in a RAM block. The size of

the RAM block is dependent on the algorithm implemented. The constants required by the algorithm are kept in an array and this array is usually implemented as a ROM: The complex arithmetic and logic computations required by the algorithm to prepare the message schedule and hash the incoming message are carried out in the hash calculation block. The control block provides necessary control signals for the padding unit, message ram, hash calculation block and constants array.

In [7] Kyu *et al*. implemented SHA-1, HAS-160 and MD5 algorithms in a single chip. These hash functions are implemented in two ways, in the first case each algorithm is implemented separately with no resource sharing; this implementation is illustrated below in Figure 2-22.

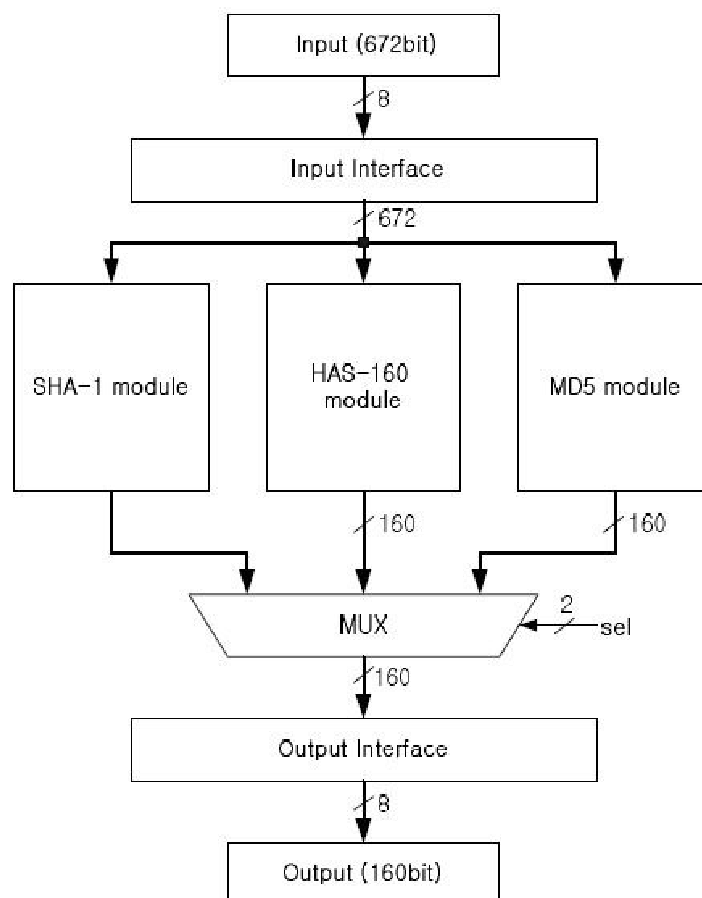

Figure 2-22 The Block Diagram of Non-Resource Sharing Design [7]

In the second case, SHA-1 and HAS-160 architectures are combined; this implementation is shown below in Figure 2-23. The designs have been implemented using Altera'sEP20K1000EBC652-3 with PCI bus interface and seen that the required logic elements are reduced by %27.

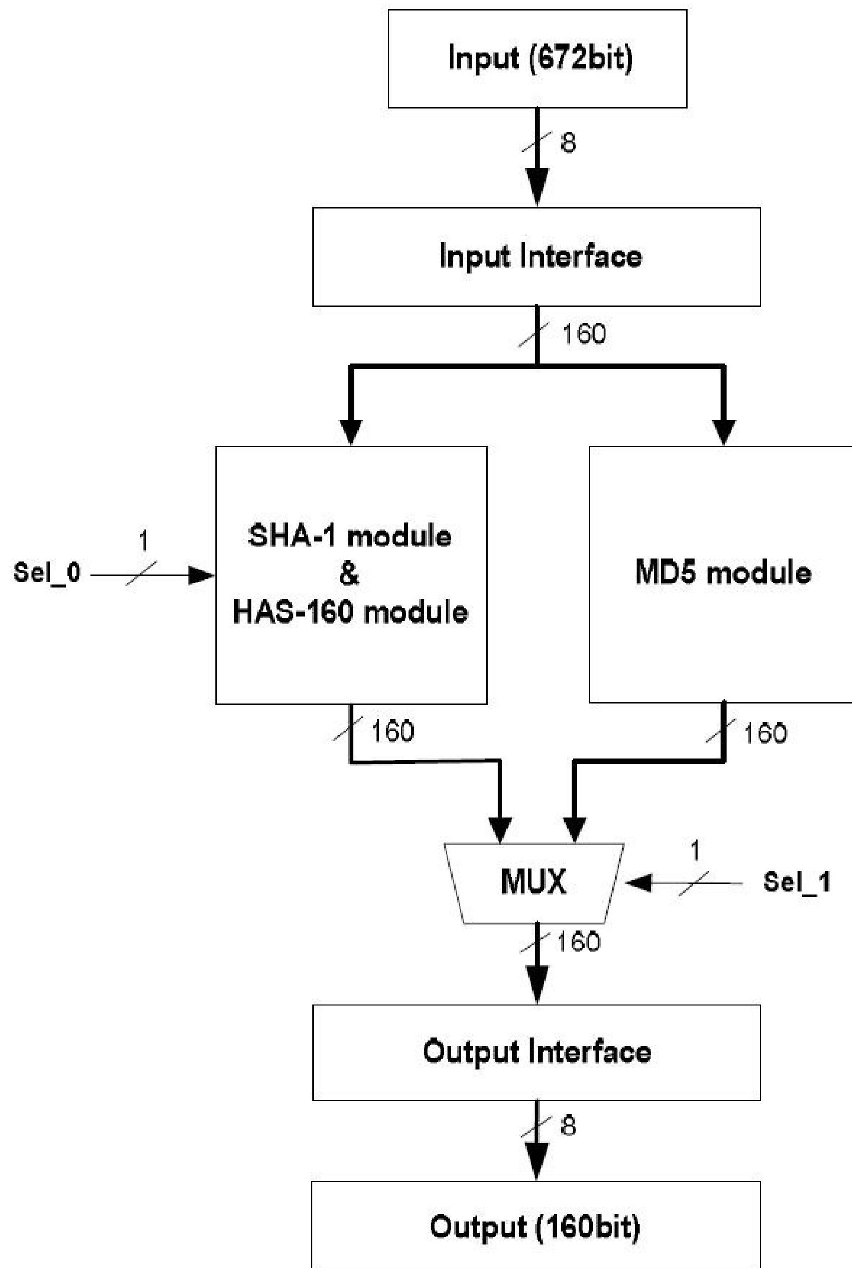

Figure 2-23 The Block Diagram of Resource Sharing Design [7]

In [8] McLoone *et al*. implemented SHA-512 and SHA-384 on a single chip. The proposed design achieves a throughput of 479 Mbps using a shift register design approach in the message scheduling part and look up tables for the constants required by the algorithms. Design is implemented on a Xilinx Virtex-E XCV600E-8 device. The design consumed 2914 CLB slices, 2 BRAMs and 141 I0Bs. The speed of the operation clock is 38 MHz. It is emphasized that the shift register design approach and the use of LUTs (Look Up Table) to store the eighty constants result in a compact and fast implementation. The shift register design approach used is illustrated in Figure 2-24.

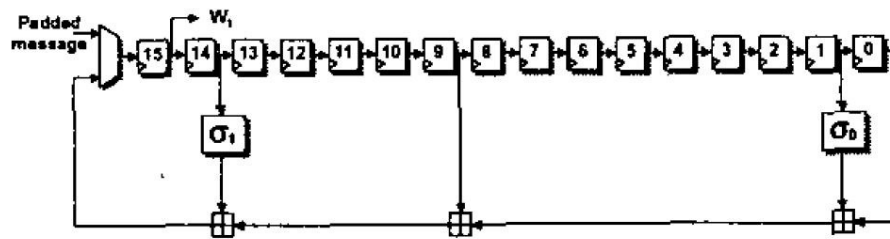

Figure 2-24 Shift Register Design Approach [8]

In [9] Grembowski *et al.* implemented SHA-1 and SHA-512 hash functions and compared the results. By using carry save adders in the hash calculation block and using the shift register approach mentioned in [17] delay and speed optimizations are done. Both algorithms are described in VHDL and implemented on Xilinx Virtex XCV-1000-6 FPGA. Throughput is found to be 670 Mbps for SHA-512 implementation and 530 Mbps for SHA-1 algorithm. As a result it is concluded that the newer algorithm SHA-512 is not only more secure than SHA-1, but also faster.

In [10], Sklavos *et al.* implemented SHA-1 and RIPEMD-160 hash functions in the same hardware module. The advantage of the proposed implementation is that it exhibits high throughput due to the pipeline technique used in the design. This pipeline technique is based on two parallel iteration loops,

left and right data paths. For SHA-1 hash calculation mode, only the left datapath is used, the right is kept idle. For the RIPEMD calculation both datapaths are used and this increases the speed of the system. The proposed left and right datapaths are illustrated below in Figure 2-25. The design is implemented on XILINX FPGA device (2VSOOfg456) and a throughput of 1339 Mbps for SHA-1 and a throughput of 1656 Mbps is observed.

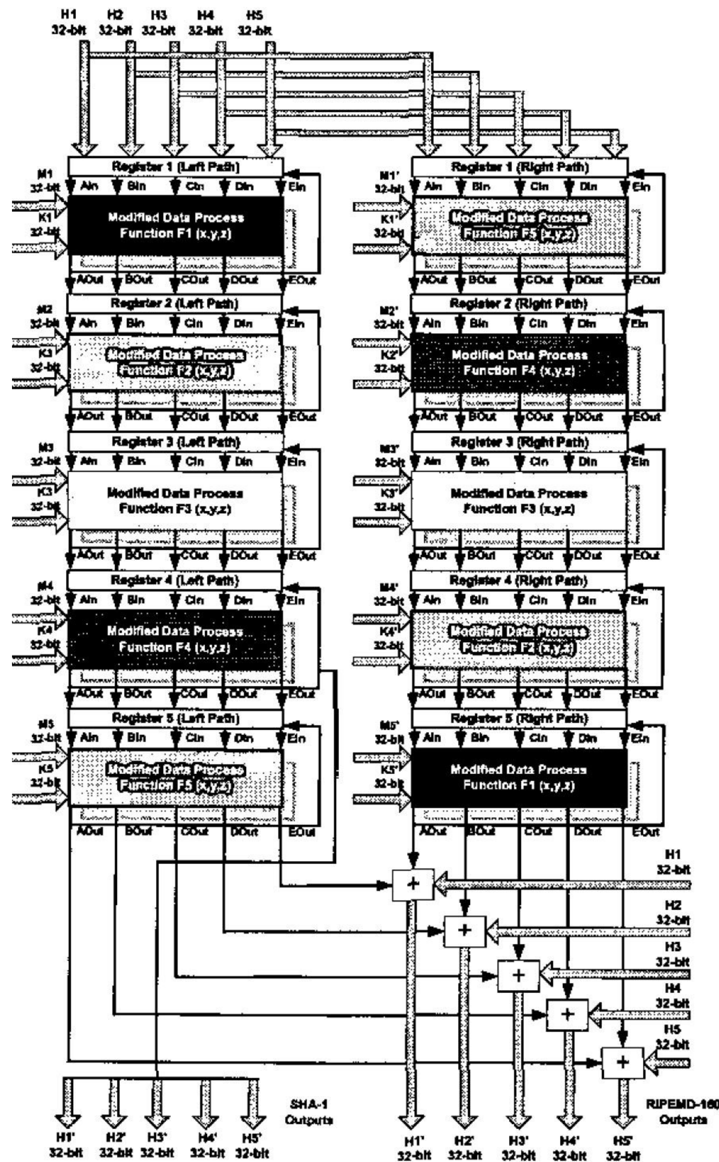

Figure 2-25 Left and Right Datapaths[10]

In [11] Sklavos *et al*. determined a common architecture for SHA-256, SHA-384 and SHA-512 hash functions and implemented these functions separately. The implementation results of the three functions are compared in the provided security level and in the performance by using hardware terms. The target device was XILINX FPGA Virtex Device (v200pq240). As a result, it is found that SHA-512 has the highest throughput; SHA-256 consumes the smallest area in the FPGA and has the best area delay product. The proposed design is illustrated below in Figure 2-26

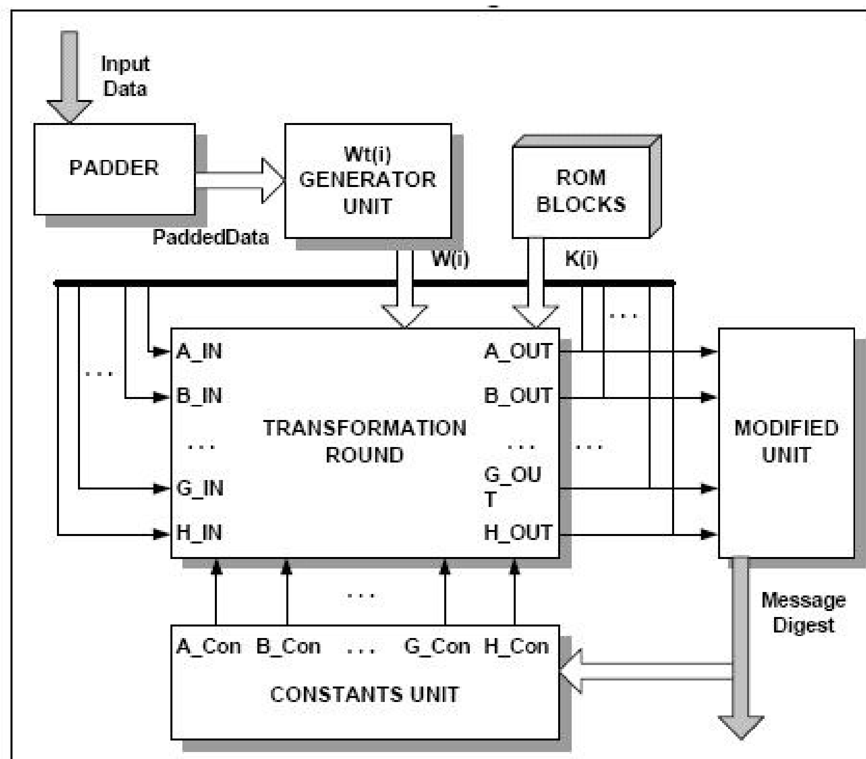

Figure 2-26 Common Architecture for SHA-256, SHA-384 and SHA-512

In [12], Michail *et al.* implemented SHA-1 hash function is in such a way that the throughput of the design is increased by %53 and the power dissipation is kept low. The proposed technique makes use of the SHA-1 hash function nature.

The idea is that except of the chaining variable a_{t-1}, the rest of the chaining variables b_{t-1}, c_{t-1}, d_{t-1} and e_{t-1} are derived directly from the variables a_{t-2}, b_{t-2}, c_{t-2}, and d_{t-2} respectively. This means consequently that also c_t, d_t and e_t can be derived directly from a_{t-2}, b_{t-2} and c_{t-2} respectively. Furthermore, due to the fact that a_t and b_t calculations require the d_{t-2} and e_{t-2} inputs respectively, which are stored in temporal registers, these calculations can be performed in parallel. Applying this method reduced the number of cycles to complete the hash calculation from 80 to 40. The core was integrated and tested on a v150bg352 FPGA device 2.8Gbps throughput is achieved.

A recent work on hash function implementations is done by Ganesh et.al.[13]. In this study a unified architecture for the hash functions MD5, SHA-1 and RIPEMD160 is proposed and the design is named as "HashChip ". The general block diagram of the HashChip is given below in Figure 2-27.

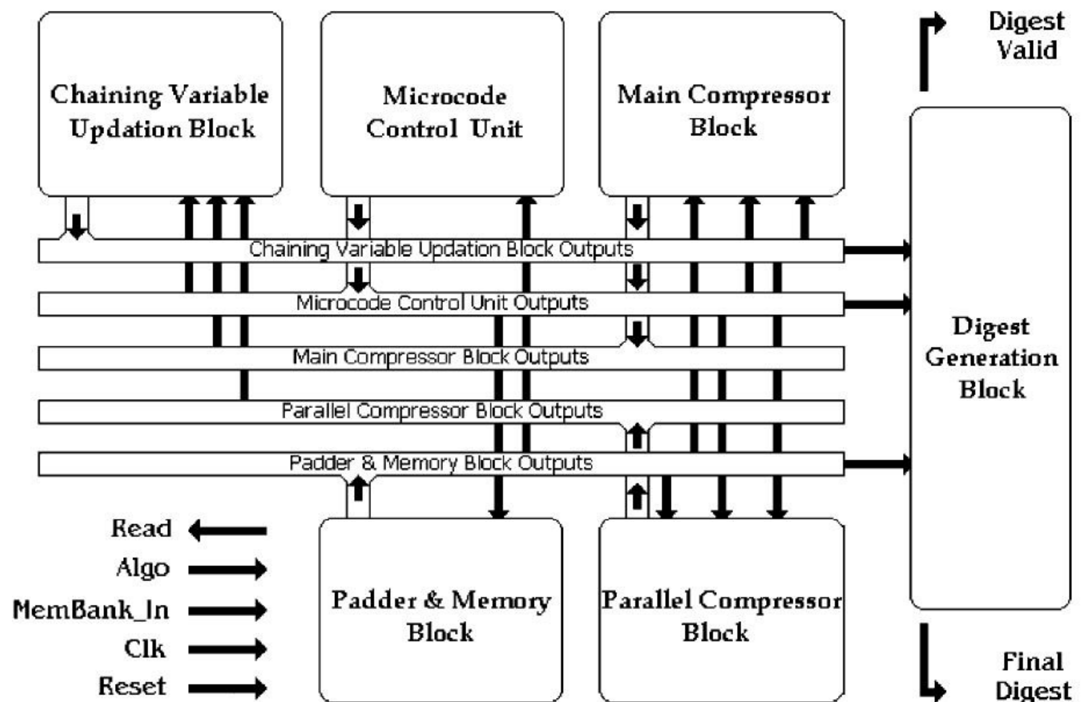

Figure 2-27 HashChip Architecture [13]

As seen above in Figure 2-27 HashChip has six main components. The padder and memory block handles the system' s interface to the external memory bank. It stores the message words and the various constants required during the iterations and also ensure that the algorithm starts processing as soon as the minimum requirements of 64 bytes are input. The chaining variable updation block updates the chaining variables at the end of the each iteration. The digest generation block ensures that the values of the chaining variables at the end of all the iterations are transferred in a proper format as the final digest value. The microcode control unit generates the necessary control signals for the datapath. The parallel and main compressor blocks implement the arithmetic and logic operations required by the algorithms. It is proposed that by removing the redundant RAM blocks the resource usage is optimized with negligible performance penalty. The critical paths are modified to accommodate lower cycle times and enable operation at higher frequencies.

The blocks described above are described using Verilog HDL at the RTL level. HashChip is implemented as part of an embedded system using Virtex II Pro and it is associated on-die PowerPC Microcontroller. The throughput of the design is compared with the existing designs and it is found that the performance of the HashChip is better than the existing implementations.

2.2.1 COMMERCIAL HASH FUNCTION IMPLEMENTATIONS

There are various hash function implementations in the market. Differ from each other according to their capabilities. Some commercial hash function implementations are listed in Table 2-8.

Cast Inc. has two hash function cores in the market, they are SHA-1 and SHA-256 hash function cores [19, 20]. Both of these cores calculates the digest of messages of any length smaller than 2^{64}- 1 bits and message lengths should be multiple of 8 bits. Bit padding operation is provided with both cores. SHA-1 calculations is completed in 82 clock cycles and SHA-256 calculation is

completed in 66 clock cycles Both cores are implemented and tested on various FPGA families and results are provided in the product datasheets. The SHA-256 and SHA-1 implementations are available as soft cores (synthesizable HDL) for ASIC technologies and as firm cores (netlist) for FPGA technologies, and include everything required for successful implementation. The functional description of the cores is as follows:

Both cores accept input message as 32-bit words and when a block of 512 bits is completed, input stream is paused and hash calculation is carried out. When processing of the 512 bit block is completed and core permits the input data to be fed again. On the final message block when the last 32-bit word is input, the core must be indicated that this is the last message word and the number of valid bytes in the last message word must be input so that padding unit knows how many bytes to pad.

HDL design house has SHA-1 function core in the market [21]. This core can accept message up to 2^{64} bits. Each 512 bit message block is processed in 80 clock cycles. The core is available as completely synthesizable VHDL or Verilog code.

Helion Technology Limited has two hash function cores in the market named as Helion Tiny Hashing core and Helion Fast Hashing core [22,23]. The first core supports SHA-1, SHA-224, SHA-256 and MD5 with or without HMAC hashing. The user can select one of these hash functions using the proper input on the core. The core is available with either 8, 16 or 32 bit data interfaces. Input message words are stored in a 512 bit block RAM in the core. After a 512 bit data block loaded, it is processed according to the algorithm selected by the micro-coded controller. This controller executes a sequence of instructions which perform a series of computations on the data block using a specially designed Arithmetic Logic Unit (ALU). The core is implemented on various Xilinx family FPGA's and the implementation results are provided in the product datasheet. The Helion Fast Hashing core has five modes of operation, these are: SHA-1 hashing, SHA-256 hashing, MD5 hashing, dual mode (SHA-1 and SHA-256) and dual

mode (SHA-1 and MD5). In all of these modes, the message is input to the core as 32-bit words. Once a 512-bit message block has been loaded, hash calculation begins. In the hash calculation process a sequence of complex arithmetic and logic functions are applied to the message words over a number of iterations. In each iteration intermediate results of the chaining variables are stored and at the end of the each block processing these are used to compute the running digest. The core is implemented on various Xilinx family FPGA's and the implementation results are provided in the product datasheet.

Aldec Inc. has an SHA-1 IP core in the market [24]. The core supports only SHA-1 hashing. 512-bit message blocks are processed in 81 clock cycles. Data is input to the core as 32-bit message words. VHDL /Verilog source code, technology-dependent EDIF and VHDL/Verilog netlists and software emulator of SHA core are delivered to the user.

Ocean Logic Pty. Ltd. has SHA-1 and SHA-256 hash function cores in the market [25, 26]. In both of the cores message is input to the core as 32-bit words. The SHA-1 calculation is completed in 81 clock cycles and SHA-256 calculation is completed in 65 clock cycles. The core is implemented on various Xilinx FPGAs also implemented as ASIC. The results of these implementations are given below in Table 2-8.

Sci-worx has a SHA-1 function core in the market [27]. The core supports only SHA-1 hashing. 512-bit message blocks are processed in 81 clock cycles. Data is input to the core as 32-bit message words. VHDL /Verilog source codes are delivered to the user.

Table 2-8 Commercial Hash Function Cores

Vendor	Supported Hash Function	Supported Platforms	Throughput	Year
CAST Inc.	SHA-1	ASIC/FPGA	6.24 Mbps/MHz	October 2007
CAST Inc.	SHA-256	ASIC/FPGA	7.75 Mbps/MHz	October 2007
HDL Design House	SHA-1	ASIC/FPGA/SoC	6.4 Mbps/MHz	December 2002
Helion Technology Limited	SHA-1 only SHA-256 only MD5 only Dual-mode (selectable SHA-1 and SHA-256) Dual-mode (selectable SHA-1 and MD5)	FPGA	SHA-1: 6.24 Mbps/MHz SHA-256: 7.75 Mbps/MHz MD5: 7.75 Mbps/MHz	July 2005
Helion Technology Limited	Supports MD5, SHA-1, SHA 224 and SHA-256 hash algorithms	FPGA	SHA-1: 0.201 Mbps/MHz SHA-224: 0.16 Mbps/MHz SHA-256: 0.16 Mbps/MHz MD5: 0.31 Mbps/MHz	July 2005
Aldec, Inc.	SHA-1	FPGA	-	2006
Ocean Logic Pty Ltd	SHA-256	FPGA/ASIC	6.325 Mbps/MHz for ASIC 0.18 u process 6.32 Mbps/MHz for Xilinx Virtex E-8 6.96 Mbps/MHz for Xilinx Virtex II-5	2005
Ocean Logic Pty Ltd	SHA-1	FPGA/ASIC	6.55 Mbps/MHz for ASIC 0.18 u process 5.5 Mbps/MHz for Xilinx Virtex E-8 6.196 Mbit/s for Xilinx Virtex II-5	2005
Sci-worx	SHA-1	FPGA	6.24 Mbps/MHz	

CHAPTER III

DESIGN OF HASH PROCESSOR

3.1 DESIGN ON FPGA

FPGAs are digital integrated circuits which contain configurable logic blocks and configurable interconnects between these logic blocks [28]. These devices can be programmed by design engineers to perform a vast variety of tasks. The "field programmable" portion of the FPGA's name refers to the fact that its programming takes place in the field [29]. This means that FPGAs are programmed in the laboratory or the function of an FPGA device which is part of a higher system can be modified easily while it is resident in the system. The general architecture of an FPGA device is shown below in Figure 3-1:

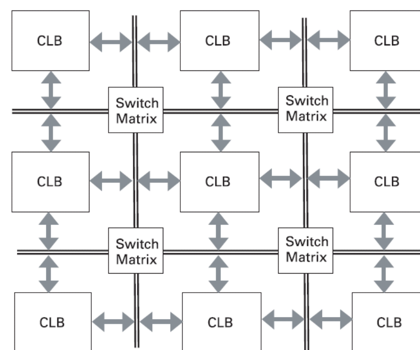

Figure 3-1 FPGA Architecture [28]

When FPGAs are first seen in the market in the mid-1980s they were mostly used to implement medium complexity state machines and very limited data processing tasks. During the early 1990s FPGAs started to be used in the telecommunications and networking areas which involve processing large blocks of data due to the increased size and complexity. Towards the end of the 1990s consumer, industrial and automotive applications are added to the areas which FPGAs are used.

FPGAs are often used to prototype ASIC designs or to provide a hardware platform on which to verify the physical implementation of new algorithms. However their low development cost and short time to market mean that they are increasingly finding their way into final products.

3.1.1 CONFIGURING FPGAs

FPGAs can be configured in two ways: in the first case case hardware description languages (HDL) are used to describe the behavior of the circuit and than this description are converted to the gate level netlist. FPGA is programmed with that netlist. This illustrated below in Figure 3-2.

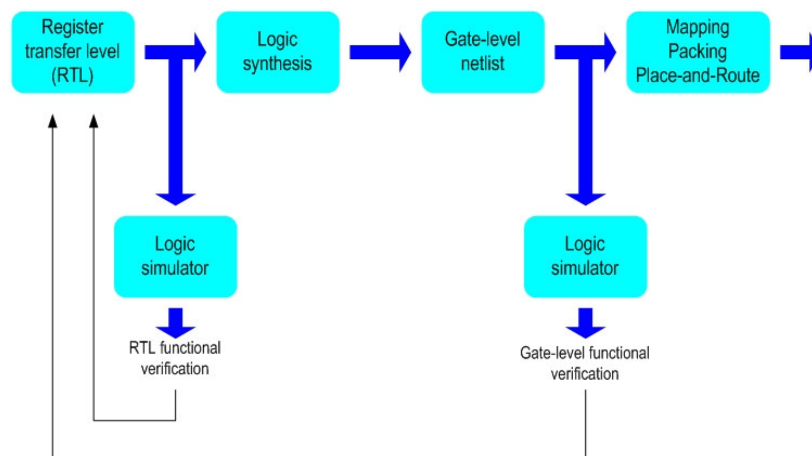

Figure 3-2 HDL Based FPGA Design Flow

In the second case the desired schematic is designed and then converted to gate level netlist and FPGA is programmed with that netlist. and FPGA is programmed with that netlist. This illustrated below in Figure 3-3.

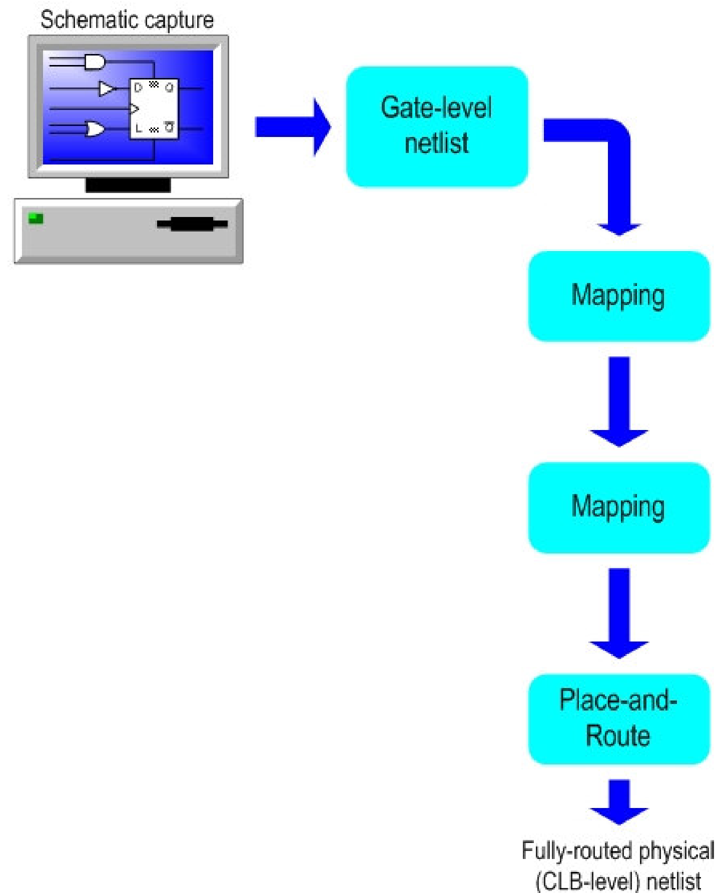

Figure 3-3 Schematic Based FPGA Design Flow

As designs grew in size and complexity, schematic based design flows ran out of stream. Visualizing, capturing, debugging, understanding and maintaining a design at the gate level of abstraction became increasingly difficult inefficient and time consuming for large designs. Thus designers preferred following HDL based design flow.

Using HDLs the functionality of a digital circuit can be described at different levels of abstraction. This illustrated below in Figure 3-4:

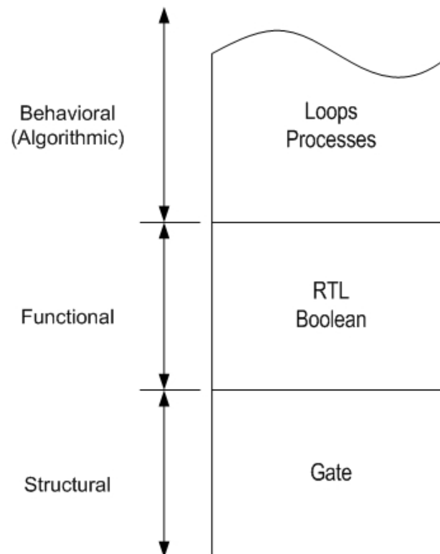

Figure 3-4 Different Levels of Abstraction

As seen in Figure 3-4, the lowest level of abstraction, the gate level, refers to the ability to describe the circuit as a netlist of primitive logic gates and functions. The functional level of abstraction is the ability to describe a function using Boolean equations. For example; with signals F, A, B and SELECT, the function of a 2:1 multiplexer can be captured as follows:

$$F = (SELECT\ \ AND\ \ A)\ \ OR\ \ (NOT(SELECT)\ \ AND\ B)$$

The functional level of abstraction also encompasses register transfer level (RTL) representations. RTL concept can be described as follows: consider a design formed from a collection of registers linked by combinational logic. These registers are often controlled by a common clock signal assuming that we have already declared two signals CLOCK and CONTROL and a set of registers REGA, REGB, REGC and REGD. Then an RTL type statement might look something like the following:

when clock rises

if CONTROL == "1"

then REGA = REGB AND REGC

else REGA = REGB OR REGD

end if;

end when;

The highest level of abstraction is known as behavioral which refers to the ability to describe the behavior of a circuit using abstract constructs like loops and processes. This also includes using algorithmic elements like adders and multipliers in equations.

In this study, VHDL is used as a hardware description language to configure the FPGA. VHDL stands for VHSIC Hardware Description Language. VHSIC is itself an abbreviation for Very High Speed Integrated Circuits, an initiative funded by the United States Department of Defense in the 1980s that led to the creation of VHDL. A fundamental motivation to use VHDL is that VHDL is a standard, technology/vendor independent language, and is therefore portable and reusable. The summary of the VHDL design flow is illustrated below in Figure 3-5.

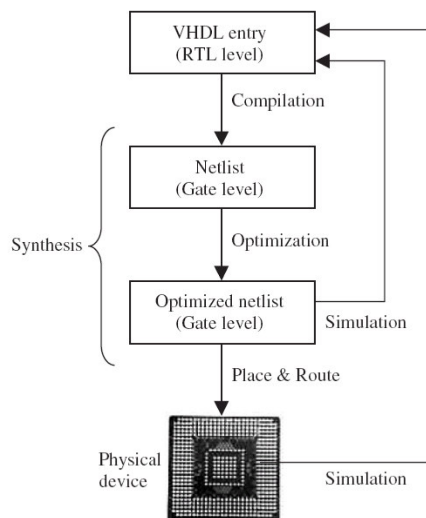

Figure 3-5 VHDL Design Flow Summary [28]

The design is started by writing the VHDL code, which is saved in a file with the extension .vhd. Then the synthesis phase comes, the first step in the synthesis process is compilation. Compilation is the conversion of the high-level VHDL language, which describes the circuit at the RTL level, into a netlist at the gate level. The second step is optimization, which is performed on the gate-level netlist for speed or for area. At this stage, the design can be simulated. Finally, the physical layout of the FPGA chip is generated by means of a place and-route (fitter) software and then FPGA is configured by a programming hardware.

3.2 HASH PROCESSOR IMPLEMENTATION

In this study SHA-1 and SHA-256 hash functions are implemented in a general processor structure. The design is fully described and captured using a hardware description language named VHDL and implemented on Xilinx FPGA. The aim is to follow all the steps in a digital hardware design flow and implement the hash functions in a processor structure rather than in classical form.

The first step in a digital hardware design process is to determine the design methodology that will be followed in order to satisfy the specifications determined. In this study, the aim is to implement the SHA-1 and SHA-256 hash functions in a processor structure. Thus as a first step, processor design on FPGA concept is examined and the design modules that are going to be implemented are determined. There are generally two types of processors: general purpose processors and dedicated processors [30]. General purpose processors such as Pentium CPU can perform different tasks under the control of software instructions. General purpose processors are used in all personal computers. Dedicated processors on the other hand are designed to perform one specific task. Dedicated processors are usually much smaller and not as complex as general purpose processors. However they are used in every smart electronic device such as TVs, cell phones, microwave ovens etc. The designed hash processor can be considered as a general purpose processor. The logic circuit of a processor can be

divided into two parts: the datapath and the control unit. The datapath is responsible for the actual execution of all data operations performed by the processor such as the addition of two numbers. Even though the datapath is capable of performing all the data operations of the processor, it can not however do it on its own. In order for the datapath to execute the operations automatically the control unit is required. The control unit is a finite state machine (FSM) because it is a machine that executes by going from one state to another that there are only a finite number of states for the machine to go. A simple block diagram of a processor is shown below in Figure 3-6.

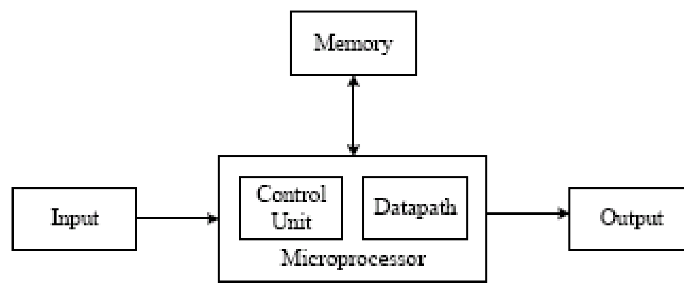

Figure 3-6 Block Diagram of a Processor

The datapath usually contains an arithmetic logic unit (ALU) and registers for temporary storage of the data. Additionally, a program memory to hold the instructions that are going to be run is a very important part of a processor. As a consequence of these it is decided that the hash processor will contain a control unit, a program memory and a datapath. The internal structure of the datapath, control unit and program memory are determined according to the properties of hash functions SHA-1 and SHA-256 and the details of the design will be given in parts 3.2.2 and 3.2.3

The design modules are designed and verified using Xilinx ISE and ModelSim. The hardware verification tests are applied on the ML402

development board [31] The implementation is done for Xilinx's Virtex4 series XC4VSX35-FF668-10 FPGA [32] which the development board includes.

3.2.1 RESOURCES USED IN THE DESIGN

The software resources used in this study is listed below in Table 3-1.

Table 3-1 Software Resources Used in the Design

Tools / Package	Usage
Xilinx ISE 7.1	Xilinx integrated synthesis and implementation tool
ModelSim XE III 6.0a	Simulation tool
Microsoft Visual Studio .NET 2003	.NET platform

VHDL description of the hash processor is written and synthesized using Xilinx ISE 7.1. This software is also used for implementation of the design and configuring the FPGA with the generated netlist. To verify the generated VHDL design description, and simulate the design, ModelSim XE III 6.0a is used.

The graphical user interface is designed in Microsoft Visual Studio .NET 2003 platform. The input text and the program that is going to be run in hash processor are sent to the hardware test platform in RS232 format using the control software developed on this platform.

The hash processor VHDL description is tested and verified on the hardware test platform. All of the hardware tools used to test the design through out this study is summarized in Table 3-2.

Table 3-2 Used Hardware for Verification

Hardware	Usage
ML402	Development kit
Xilinx Platform Cable USB	Programmer over JTAG port
Test Computer	User interface software runs on

ML402 Development Kit [31] constitutes the hash processor' s hardware test platform. ML402 development kit includes Xilinx Virtex-4 series XC4VSX35-FF668-10 FPGA. The FPGA is configured using the Xilinx's Platform Cable USB [33]. This programmer is high-speed download cable that configures or programs all Xilinx FPGA, CPLD, ISP PROM, and System ACE MPM devices. The test hardware specifications are given in Appendix C. Hardware test platform setup is presented in Chapter 4.

3.2.2 HASH PROCESSOR ARCHITECTURE AND INSTRUCTION SET

In this study, for the implementation of the hash functions SHA-1 and SHA-256, a processor structure is proposed. When determining the modules that will constitute the hash processor, properties of SHA-1 and SHA-256 hash functions are taken into account.

Hash processor designed in this study is composed of the following modules listed:

- Control Unit
- Datapath
  - Message Expansion Block
  - Constants Rom
  - Register File
  - ALU
- Program Memory
- UART Interface

Controller, datapath and program memory are the main blocks of a processor. The datapath generally contains a register file and an arithmetic logic unit in order to handle the arithmetic operations required. However, in the proposed architecture, datapath includes two more modules, a constants rom block and a message expansion block which are designed in order to satisfy some special

requirements of the hash functions SHA-1 and SHA-256. The general block diagram of the hash processor is shown below in Figure 3-7.

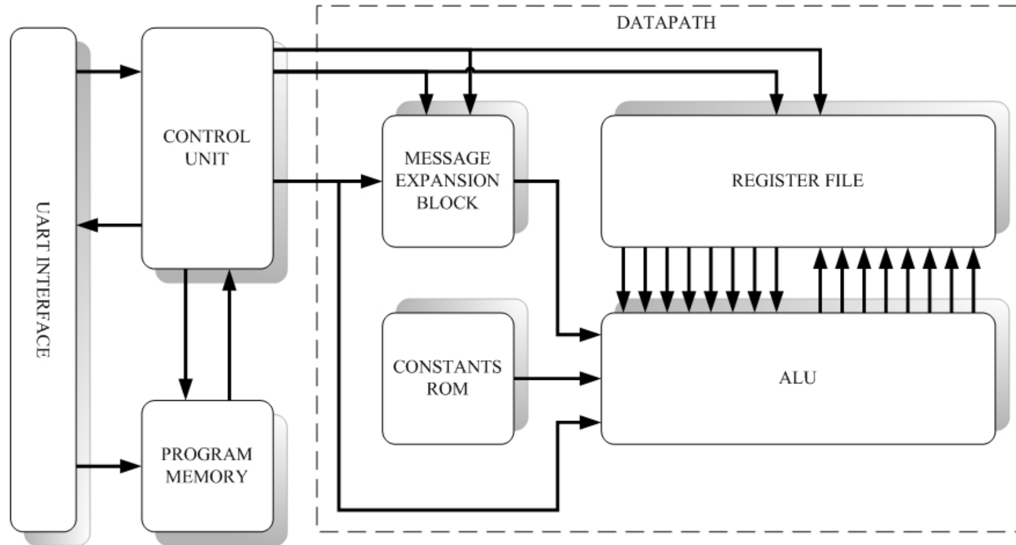

Figure 3-7 Hash Processor General Block Diagram

After determining the modules and interaction between these modules, the first thing that is going to be done is to examine the hash functions in detail to develop special instructions. These instructions are given in Table 3-3:

Table 3-3 Hash Processor Instructions

Instruction Mnemonic & Opcode	Machine Code	Brief Description
LDA loc 0000	000000000000000000000000000xxxxx	Load accumulator with the contents of the memory location
AND loc 1001	000000000000000000000001001xxxxx	AND accumulator with the contents of the memory location
ADD loc 0010	000000000000000000000000010xxxxx	ADD the contents of the memory location

Instruction Mnemonic & Opcode	**Machine Code**	**Brief Description**
		to accumulator
SUB loc 0011	000000000000000000000000011xxxxx	SUB the contents of the memory location from accumulator
JMPP addr 0110	000000000000000000000000110xxxxx	Jump to address if the content of the acc is positive
STA loc 1000	000000000000000000000001000xxxxx	Store accumulator to memory location
JMPZ addr 1111	000000000000000000000001111xxxxx	Jump to address if the content of the acc is zero
SHA1 0001	000000000000000000000000001xxxxx	generates the necessary control signals for the datapath to make one step SHA-1 calculation
SHA2 1010	000000000000000000000001010xxxxx	generates the necessary control signals fort he datapath to make one step SHA-256 calculation
SRGF1 1011	000000000000000000000001011xxxxx	stores the intermediate hash values of the SHA1 algorithm to the register file
SRGF2 1101	000000000000000000000001101xxxxx	stores the intermediate hash values of the SHA-256 algorithm to the register file
RRGF1 1100	000000000000000000000001100xxxxx	reads the intermediate hash values of the SHA-1 algorithm from the register file
RRGF2 1110	000000000000000000000001110xxxxx	reads the intermediate hash values of the SHA-256 algorithm

Instruction Mnemonic & Opcode	Machine Code	Brief Description
		from the register file.
HALT 0111	00000000000000000000000111xxxxx	terminates the execution of the processor

The instructions given in Table 3-3 are determined for SHA-1 and SHA-256 hash functions as a starting point and can be extended for other hash functions easily.

3.2.3 HASH PROCESSOR MODULES

3.2.3.1 CONTROL UNIT

Control unit is the main controller of the hash processor. It is mainly a finite state machine that generates the necessary control signals for the datapath. As an addition, the control unit has UART interface that enables communication with a PC' s serial port. The input output signals of the control unit are shown below in Table 3-4:

Table 3-4 Input Output Signals of the Control Unit

Port name	Direction	Description
clock	input	100 MHz clock signal
reset	input	global reset signal
input	input	32-bit input to the controller
round	output	7-bit output for the datapath. Holds the hash operation round number
RFWe	output	Single bit output for the datapath. Enables datapath to write to the registerfile
Mwe	output	Single bit output for the datapath. Enables

Port name	Direction	Description
		datapath to write to the message ram.
ALUe	output	Single bit output for the datapath. Enables the ALU in the datapath
RFr1e	output	Single bit output for the datapath. enables the first register in the register file to read its content
RFr2e	output	Single bit output for the datapath. enables the second register in the register file to read its content
RFr3e	output	Single bit output for the datapath. enables the third register in the register file to read its content
RFr4e	output	Single bit output for the datapath. enables the fourth register in the register file to read its content
RFr5e	output	Single bit output for the datapath. enables the fifth register in the register file to read its content
RFr6e	output	Single bit output for the datapath. enables the sixth register in the register file to read its content
RFr7e	output	Single bit output for the datapath. enables the seventh register in the register file to read its content
RFr8e	output	Single bit output for the datapath. enables the eight‘th register in the register file to read its content
RFr1wa	output	4 bit output for the datapath. Holds the address of the register to write to the first input of the registerfile
RFr2wa	output	4 bit output for the datapath. Holds the address of the register to write to the second input of the registerfile
RFr3wa	output	4 bit output for the datapath. Holds the address of the register to write to the third input of the registerfile
RFr4wa	output	4 bit output for the datapath. Holds the address of the register to write to the fourth input of the registerfile
RFr5wa	output	4 bit output for the datapath. Holds the address of the register to write to the fifth input of the registerfile

Port name	Direction	Description
RFr6wa	output	4 bit output for the datapath. Holds the address of the register to write to the sixth input of the registerfile
RFr7wa	output	4 bit output for the datapath. Holds the address of the register to write to the seventh input of the registerfile
RFr8wa	output	4 bit output for the datapath. Holds the address of the register to write to the eight'th input of the registerfile
RFr1a	output	4 bit output for the datapath. Holds the address of the register to read the contents to the first output of the registerfile
RFr2a	output	4 bit output for the datapath. Holds the address of the register to read the contents to the second output of the registerfile
RFr3a	output	4 bit output for the datapath. Holds the address of the register to read the contents to the third output of the registerfile
RFr4a	output	4 bit output for the datapath. Holds the address of the register to read the contents to the fourth output of the registerfile
RFr5a	output	4 bit output for the datapath. Holds the address of the register to read the contents to the fifth output of the registerfile
RFr6a	output	4 bit output for the datapath. Holds the address of the register to read the contents to the sixth output of the registerfile
RFr7a	output	4 bit output for the datapath. Holds the address of the register to read the contents to the seventh output of the registerfile
RFr8a	output	4 bit output for the datapath. Holds the address of the register to read the contents to the eight'th output of the registerfile
ALUsel	output	3 bit output for the datapath. Tells the ALU which arithmetic operation to carry out
A	output	32-bit output for the datapath and program memory. Holds the value of the accumulator.
memory_data	output	32-bit output for the datapath. Holds the content of the addressed location of the program memory.

Port name	Direction	Description
MemWr	output	Single bit output for the program memory. Enables the program memory for write operation
Memory_address	output	5-bit output for the program memory. Holds the address of the memory location to read from or to write to.

Control unit includes two 32 bit registers, accumulator A and, program counter PC: The operation of the controller is best explained by describing the states of the controller in detail. The states of the controller are:

- s_fill: This is the controller's beginning state. This state waits for the serial buffer to be filled with data. When the serial buffer is filled with data, the program memory is filled with the data received. This operation is handled by the sub states in this state. The sub states are:
    - s_init: This is the initial state of the filling operation. If the serial buffer is filled, the serial data is copied to the buffer "data" and controller passes to the next state s_fill_1, otherwise it waits in this state. The value of the 5 bit variable instr_count is set to "00000".
    - s_fill_1: In this state last 32 bits of the serially received data is copied to the accumulator. The address of the program memory is set to the value of the instr_count. Next state from this state is s_fill_2.
    - s_fill_2: In this state, MemWr signal is set in order to store the value in the accumulator to the program memory. Next state is s-fill_3.
    - s_fill_3: In this state, MemWr signal is deserted in order to avoid uncontrolled write operations to the memory. The variable instr_count is incremented. The contents of the buffer "data" is shifted to right by 32 bits. If the value of the variable instr_count reaches to "11111" next state is s_end, else next state is s_fill_1.
    - s_end: In this sate, only the next state is determined to be s_start.

- s_start: In this state controller begins its normal operation. The address of the program memory is set to the value of the program counter. Next state is the s_fetch state.
- s_fetch: In this state the value of the instruction register is set to the value read from the program memory. The value of the program counter is incremented. Next state is the s_decode state.
- s_decode: In this state, address of the program memory is set to the first five bits of the instruction register. Last four bits of the instruction register defines the next state to go. These last four bits are the opcodes and are shown below in Table 3-5.

Table 3-5 Opcodes

opcode	state
0000	load
0001	s_sha1
0010	s_add
0011	s_sub
0100	s_input
0101	s_sha1_out
0110	s_jpos
0111	s_halt
1000	s_store
1001	s_and
1010	s_sha2
1011	s_store_regf_1
1100	s_read_regf_1
1101	s_store_regf_2
1110	s_read_regf_2
1111	s_jz

- s_load: In this state accumulator is load with the value read from the program memory. Next state is the state s_start.

- s_add:In this state the value in the accumulator and the value read from program memory is added and the result is stored in the accumulator. Next state is the state s_start.
- s_and: In this state the value in the accumulator is anded with the value read from the program memory and the result is stored back in the accumulator. Next state is the state s_start.
- s_sub: In this state the value read from the program memory is subtracted from the value in the accumulator and the result is stored back in the accumulator. Next state is the state s_start.
- s_input: In this state the value input to the controller is stored in the accumulator. Next state is the state s_start.
- s_store: In this state the value in the accumulator is stored in the program memory. Next state is the state s_store2.
- s_store2: In this state the value of the signal Memwr which is set to '1' in the previous state is set to '0' Next state is s_start.
- s_jpos: In this state, if the value in the accumulator is positive, program counter is set to the first five bits of the instruction register, thus jumped to the memory location pointed by first five bits of the instruction register. Next state is the state s_start.
- s_jz: In this state, if the value in the accumulator is zero, program counter is set to the first five bits of the instruction register, thus jumped to the memory location pointed by first five bits of the instruction register. Next state is the state s_start.
- s_halt: In this state the execution of the controller is terminated.
- s_sha1: In this state, the control signals necessary for the datapath to execute one round of SHA-1 operation are generated. Next state is the state s_start.
- s_sha2: In this state, the control signals necessary for the datapath to execute one round of SHA-2 operation are generated. Next state is the state s_start.

- s_store_regf_1: In this state the chaining variables related to the SHA-1 hash function are stored in the register file. Next state is the state s_start.
- s_store_regf_2: In this state the chaining variables related to the SHA-256 hash function are stored in the register file. Next state is the state s_start.
- s_read_regf_1: In this state the values of the chaining variables related to the SHA-1 hash function are read from the register file. Next state is the state s_start.
- s_read_regf_2: In this state the values of the chaining variables related to the SHA-256 hash function are read from the register file. Next state is the state s_start.

After these explanations the state diagram of the controller is shown below in Figure 3-8.

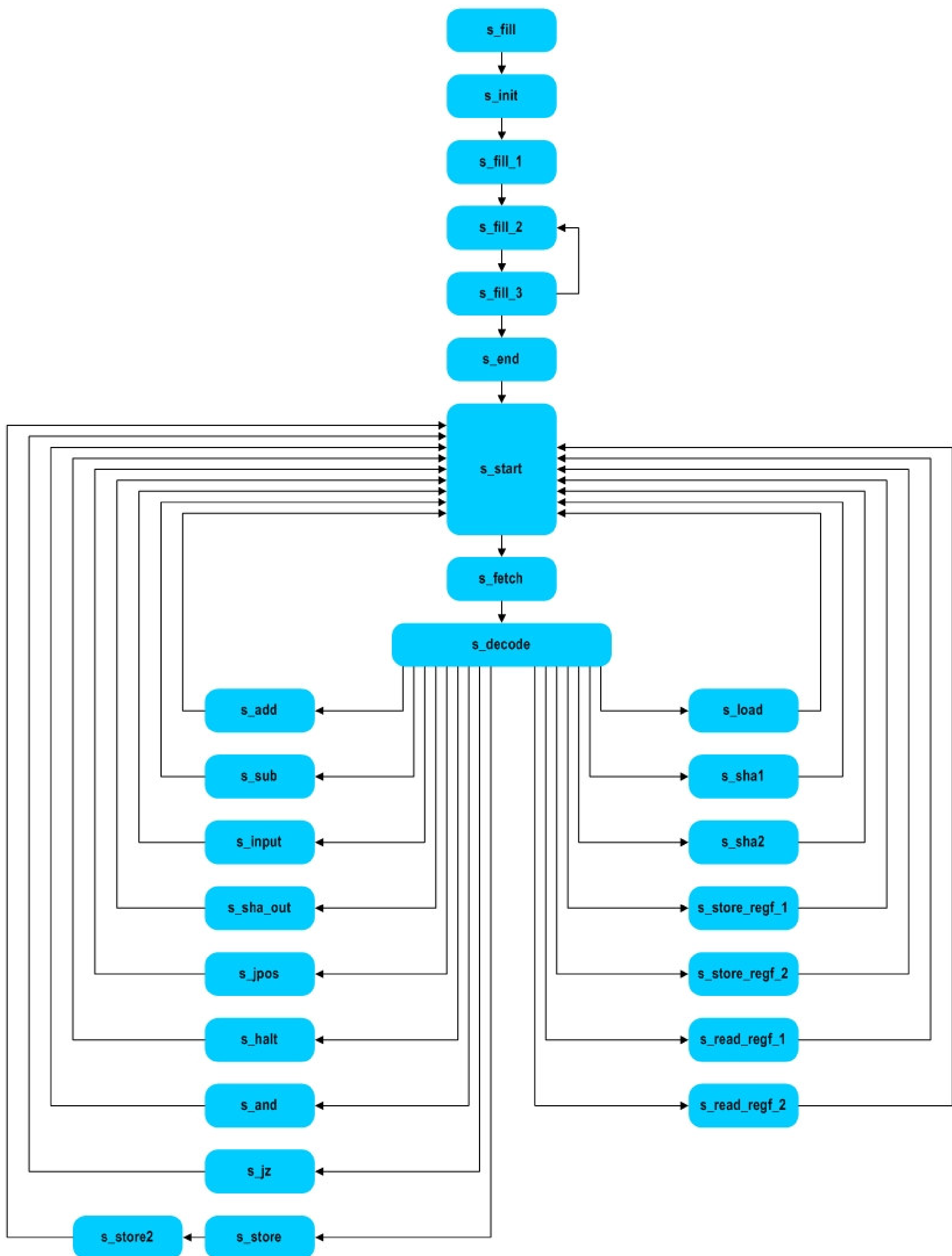

Figure 3-8 Controller State Diagram

3.2.3.2 PROGRAM MEMORY

Program memory is designed as a 32x32 RAM to hold the program instructions. The input output signals of the program memory are shown below in Table 3-6.

Table 3-6 Input Output Signals of the Program Memory

Port Name	Direction	Description
clk	input	Clock input
we	input	Single bit input used for enabling write operation
a	input	5-bit input used for addressing the RAM
di	input	32-bit RAM input
do	output	32-bit RAM output

Program memory is synthesized as 32x32 BRAM. The write operation is synchronized to the rising edge of the clock input but read operations independent of the clock.

3.2.3.3 DATAPATH

The general block diagram of the datatapath is shown below in Figure 3-9.

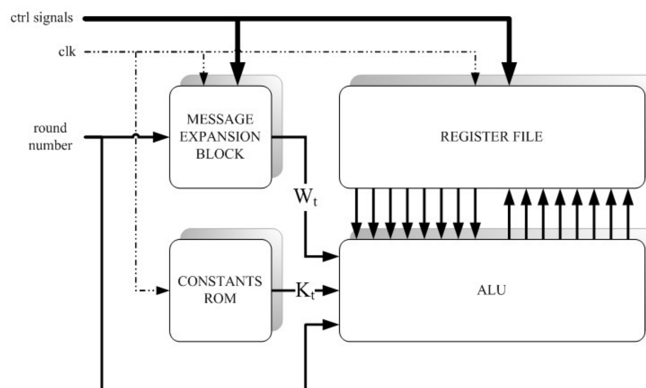

Figure 3-9 Datapath Architecture

Datapath is composed of the modules message expansion block, constants ROM, register file and ALU. The connections between these modules are described in the following sections in details. The input output signals and the definitions of these signals are also given.

3.2.3.3.1 MESSAGE EXPANSION BLOCK

This block is responsible of storing the incoming 512-bit message block and preparing the message schedule. The input output signals of the message block are shown below in Table 3-7.

Table 3-7 Input Output Signals of the Message Expansion Block

Port Name	Direction	Description
clock	input	Clock input
reset	input	Single bit global reset input
we	input	5-bit input used for addressing the RAM
round	input	7-bit input that holds the hash round number
SEL	input	3-bit input holds the value that determines which functions to use in order to prepare the message schedule
di	input	32-bit RAM input
do1	output	32 bit RAM output

Message expansion block is composed of a 80x32 RAM to store the incoming 512 bit message block and some processes to generate the message schedule. For the SHA-1 hash function, 80 message words are quired, for the SHA-256 hash function 64 message words are required, so the size of the RAM is determined to be 80x32. In the first process according to the SEL signal and the round number addresses of the message words that will be used to prepare the message schedule are generated. In the next two processes message schedule is prepared according to the selected hash function. Message expansion block is synthesized as 4 32x80 BRAMs and some extra logic.

3.2.3.3.2 CONSTANTS ROM

This module holds the constant values required by the hash function SHA-1 and SHA-256. There are 64 different constants used in the 64 rounds of the SHA-256 hash function where there are four constants used in the 4x20 rounds of the SHA-1 hash function. Thus this rom block is 80x64 bits wide. Last 32 bits of a ROM entry holds one SHA-256 constant and first 32 bits hold one SHA-1 constant. The input output signals of the ROM block are shown below in Table 3-8.

Table 3-8 Input Output Signals of the ROM Block

Port Name	Direction	Description
clock	input	Clock input
reset	input	Single bit global reset input
SEL	input	3-bit input that holds the value that determines which hash function' s constants to read.
address	input	7 bit input that holds the rom address
K	output	32 bit rom output

ROM block is synthesized as 80x32 ROM structure.

3.2.3.3.3 REGISTER FILE

Register file is the module that holds the chaining variables of the hash functions SHA-1 and SHA-256. There are five chaining variables for SHA-1 hash function and eight chaining variables for the SHA-256 hash function. There are total 16 registers in the register file. Three registers are intentionally left blank in order to be used for accumulator or memory data coming from the program memory when needed. The input and output signals of the register file are shown below in Table 3-9.

Table 3-9 Input Output Signals of the Register File

Port name	Direction	Description
clock	input	clock signal

Port name	Direction	Description
reset	input	global reset signal
RFWe	input	Single bit input. Enables the register file for write operation
RFr1e	input	Single bit input. Enables the read operation from the first output of the register file
RFr2e	input	Single bit input. Enables the read operation from the second output of the register file
RFr3e	input	Single bit input. Enables the read operation from the third output of the register file
RFr4e	input	Single bit input. Enables the read operation from the fourth output of the register file
RFr5e	input	Single bit input. Enables the read operation from the fifth output of the register file
RFr6e	input	Single bit input. Enables the read operation from the sixth output of the register file
RFr7e	input	Single bit input. Enables the read operation from the seventh output of the register file
RFr8e	input	Single bit input. Enables the read operation from the eight' th output of the register file
RFr1wa	input	4-bit input. Determines the address of the register to which the value in the first input will be written
RFr2wa	input	4-bit input. Determines the address of the register to which the value in the second input will be written
RFr3wa	input	4-bit input. Determines the address of the register to which the value in the third input will be written
RFr4wa	input	4-bit input. Determines the address of the register to which the value in the fourth input will be written
RFr5wa	input	4-bit input. Determines the address of the register to which the value in the fifth input will be written
RFr6wa	input	4-bit input. Determines the address of the register to which the value in the sixth input will be written
RFr7wa	input	4-bit input. Determines the address of the register to which the value in the seventh input will be written

Port name	Direction	Description
RFr8wa	input	4-bit input. Determines the address of the register to which the value in the eight' th input will be written
RFr1a	input	4-bit input. Determines the address of the register from which the value will be read to the first output of the register file
RFr2a	input	4-bit input. Determines the address of the register from which the value will be read to the second output of the register file
RFr3a	input	4-bit input. Determines the address of the register from which the value will be read to the third output of the register file
RFr4a	input	4-bit input. Determines the address of the register from which the value will be read to the fourth output of the register file
RFr5a	input	4-bit input. Determines the address of the register from which the value will be read to the fifth output of the register file
RFr6a	input	4-bit input. Determines the address of the register from which the value will be read to the sixth output of the register file
RFr7a	input	4-bit input. Determines the address of the register from which the value will be read to the seventh output of the register file
RFr8a	input	4-bit input. Determines the address of the register from which the value will be read to the eight' th output of the register file
RFin1	input	32-bit input to the register file
RFin2	input	32-bit input to the register file
RFin3	input	32-bit input to the register file
RFin4	input	32-bit input to the register file
RFin5	input	32-bit input to the register file
RFin6	input	32-bit input to the register file

Port name	Direction	Description
RFin7	input	32-bit input to the register file
RFin8	input	32-bit input to the register file
RFr1	output	32-bit output of the register file
RFr2	output	32-bit output of the register file
RFr3	output	32-bit output of the register file
RFr4	output	32-bit output of the register file
RFr5	output	32-bit output of the register file
RFr6	output	32-bit output of the register file
RFr7	output	32-bit output of the register file
RFr8	output	32-bit output of the register file

The register file works as follows, at the beginning of each hash round the value of the chaining variables are read from the register file, then at the end of the each hash round, the value of the chaining variables is written to the corresponding registers in the register file. This operation continues until hash computation ends.

3.2.3.3.4 ALU

Arithmetic logic unit is the module which handles all the arithmetic, logic calculations instructed by the controller. This part is the hearth of the hash calculation process. The logic functions which are specific to the hash functions SHA-1 and SHA-256 are all implemented in this module. The input output signals of this module are shown below in Table 3-10.

Table 3-10 Input Output Signals of the ALU

Port Name	Direction	Description
clock	input	Clock input
reset	input	Single bit global reset input
ALUe	input	Single bit input that enables the ALU
round	input	7 bit input that holds the number of the sha round
SEL	input	3-bit input that holds the value that determines which arithmetic logic computation to execute
ACC_in	input	32-bit input which holds the accumulator value.
Mem_data	input	32-bit input that holds the values read from the program memory
A	input	First chaining variable for SHA-1 or SHA-256 hash function
B	input	Second chaining variable for SHA-1 or SHA-256 hash function
C	input	Third chaining variable for SHA-1 or SHA-256 hash function
D	input	Fourth chaining variable for SHA-1 or SHA-256 hash function
E	input	Fifth chaining variable for SHA-1 or SHA-256 hash function
F	input	Sixth chaining variable for SHA-256 hash function
G	input	Seventh chaining variable for SHA-256 hash function
H	input	Eight' th chaining variable for SHA-256 hash function
K	input	Constant value read from Constants ROM
W	input	Message word coming from the message computation block
ALU_sign	output	The sign of the result of the operation executed by ALU (if positive '1', if zero '0')
ALU_out	output	The output of the operation executed by ALU
A_out	output	First chaining variable after one round hash calculation
B_out	output	Second chaining variable after one round hash

Port Name	Direction	Description
		calculation
C_out	output	Third chaining variable after one round hash calculation
D_out	output	Fourth chaining variable after one round hash calculation
E_out	output	Fifth chaining variable after one round hash calculation
F_out	output	Sixth chaining variable after one round hash calculation
G_out	output	Seventh chaining variable after one round hash calculation
H_out	output	Eight' th chaining variable after one round hash calculation

ALU selects which calculation to execute according to the instruction decoded by the control unit and input to the ALU as the SEL signal. The operation selection according to the input SEL is shown below in Table 3-11.

Table 3-11 ALU Operation Selection

SEL	Operation
000	One round SHA-1 operation
001	One round SHA-256 operation
010	Addition operation
011	Subtraction operation
100	And operation
101	Or operation

3.2.3.4 UART MODULE

The UART module is composed of UART receiver, UART baud generator and UART transmitter sub modules. The UART module provides a serial interface between the control unit and the external environment via

sdata_in pin. Serial data sent in RS-232 format is received at each positive edge of the clock and received bytes are put into the receive shift register.

The UART receive module is used to get the instructions to be executed by the controller and the 512 bit message blocks from the user.

The UART baud generator module supports six different baud rates. They are shown below in Table 3-12. In this study 38400 Hz baud rate is selected for communication.

Table 3-12 UART Baud Rate Selection Table

Baud Rate Selection Register	Generated Clock Frequency
“0110”	38400 Hz
“0101”	19200 Hz
“0100”	9600 Hz
“0011”	4800 Hz
“0010”	2400 Hz
“0001”	1200 Hz

CHAPTER IV

HARDWARE REALIZATION OF HASH PROCESSOR

5.1 HASH PROCESSOR OVER AN FPGA

In this study the aim is to implement the designed hash function processor on FPGA. For this purpose Xilinx ML402 Evaluation Platform is used. This platform is a development kit that has several features and includes the following components (the numbers represented in parenthesis in the below list are the numbers which indicate the components on Figure 5-1 and Figure 5-2 given below.):

- Virtex-4 FPGA XC4VSX35-FF668-10 (1)
- 64 MB DDR SDRAM, 32-bit interface running up to 266 MHz data rate (2)
- One differential clock input pair and differential clock output pair with SMA connectors (3)
- One 100 MHz clock oscillator (socketed) plus one extra open 3.3V clock oscillator socket (4)
- General purpose DIP switches (ML401/ML402 platform), LEDs, and push buttons (6, 7, 8, 9)
- Expansion header with 32 single-ended I/O, 16 LVDS capable differential pairs (10)

- 14 spare I/O's shared with buttons and LEDs, power, JTAG chain expansion capability, and IIC bus expansion (10)
- Stereo AC97 audio codec with line-in, line-out, 50-mW headphone, and microphone-in (mono) jacks (11)
- RS-232 serial port (12)
- 16-character x 2-line LCD display (13)
- 4 Kb IIC EEPROM (14)
- VGA output with 50 MHz / 24-bit video DAC (140 MHz on ML402/ML403) (15)
- PS/2 mouse and keyboard connectors (16)
- System ACE™ CompactFlash configuration controller with Type I/II CompactFlash connector (17)
- ZBT synchronous SRAM (9 Mb) on 32-bit data bus with four parity bits) (18)
- Intel StrataFlash (or compatible) linear flash chips (8 MB) (19)
- 10/100/1000 tri-speed Ethernet PHY transceiver (21)
- USB interface chip (Cypress CY7C67300) with host and peripheral ports (22)
- Xilinx XC95144XL CPLD to allow linear flash chips to be used for FPGA configuration (20)
- Xilinx XCF32P Platform Flash configuration storage device (23)
- JTAG configuration port for use with Parallel Cable III or Parallel Cable IV cable (24)
- Onboard power supplies for all necessary voltages (25)
- 5V @ 3A AC adapter (26)
- Power indicator LED (27)

The front side of the development kit is shown below in Figure 5-1.

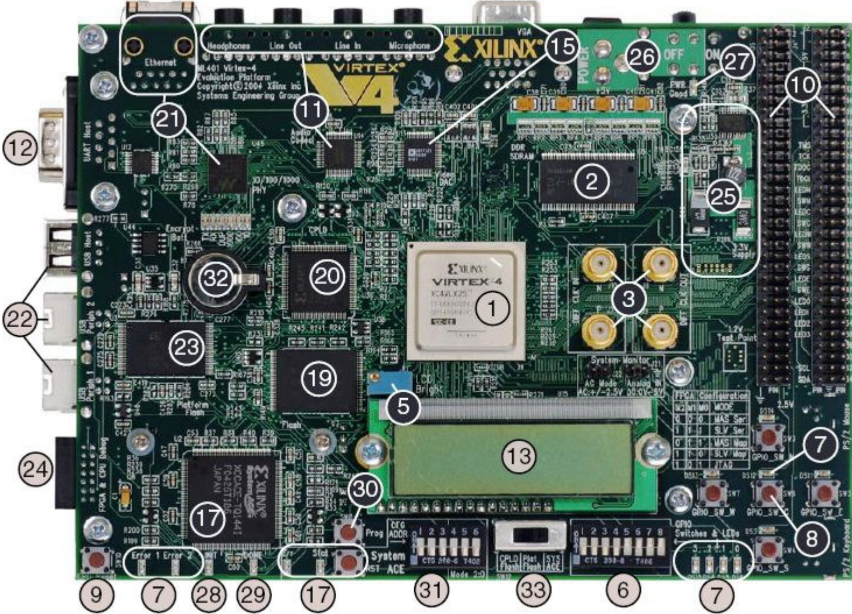

Figure 5-1 Xilinx ML402 Evaluation Platform Front Side

The back side of the development kit is shown below in Figure 5-2.

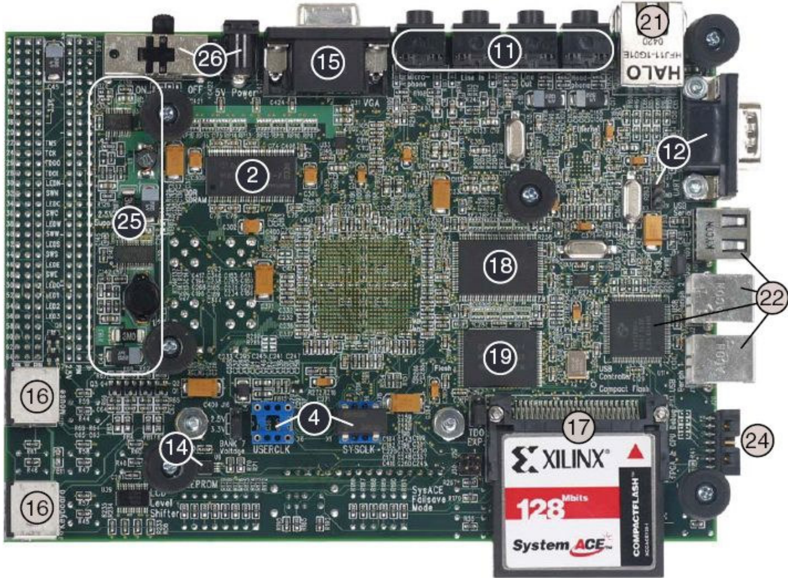

Figure 5-2 Xilinx ML402 Evaluation Platform Back Side

Xilinx ML402 contains Xilinx's Virtex4 series XC4VSX35 FPGA [20]. Virtex-4 FPGAs deliver breakthrough performance at the lowest cost and offer a compelling alternative to ASICs. The development board also has RS-232 serial port which makes communication with peripherals possible.

Hash processor is fully described using VHDL on Xilinx ISE software. Target FPGA also belongs to the same company. This is an advantage since Xilinx ISE software provides full support for all the code-to-FPGA processes for Xilinx FPGAs. The steps in the implementation process are described below:

1. Synthesis: In the synthesis process the syntax of the design is checked and the written VHDL descriptions are converted to the common constructs on the FPGA such as multiplexers, flip flops, BRAMs etc.
2. Implement design: Before implementation, the constraint file is written to define hardware I/O connections. The implementation constraints file includes timing constraints, package pin assignments and area constraints. Implementing the design means translating, mapping, placement and routing of the design into the targeted Xilinx device. In this process, logical design file generated in the synthesis process, is converted into a native circuit description (NCD file). This file contains hierarchical components used to develop the design and the Xilinx primitives.
3. Generate programming file: In order to generate programming file, the design should have been implemented for the selected FPGA device. This process generates the ".bit" file required to program the FPGA.
4. Configure the device: This process is the process where the FPGA is programmed. FPGA is programmed using Xilinx's Platform Cable USB [21].

As described above hash processor descriptions are synthesized and implemented in the Xilinx ISE software platform. At the end of the synthesis process

behavioral simulations of the design is carried out. After the implementation process timing simulations are done, programming file is generated and the device is configured using Xilinx's Platform Cable USB. After the device is configured, it is ready to perform hardware tests of the design. The device utilization summary of the design after implementation is given below in Table 5-1.

Table 5-1 Device Utilization Summary for Hash Processor VHDL Code

Logic Utilization	Used	Available	Utilization
Number of Slices	2494	15360	16%
Number of Slices Flip Flops	2793	30720	9%
Number of 4 input LUTs	3872	30720	12%
Number of bonded IOBs:	350	450	77%
Number of FIFO16/RAMB16s:	4	192	2%

5.2 TEST AND VERIFICATION METHODOLOGY

The proposed hash function processor is tested and verified in two stages. First stage is the verification on the software platform. Second stage is the verification on the hardware platform. In both stages the test vectors which are published by NIST are used as input messages and the outputs are compared with the actual results. For the random inputs, the design is verified by using a software named "Advanced Hash Calculator" which is available on internet for free. The hash value of the random input is calculated by this software and by the proposed design then the results are compared. The snapshot of the "Advanced Hash Calculator Software" is shown below in Figure 5-3.

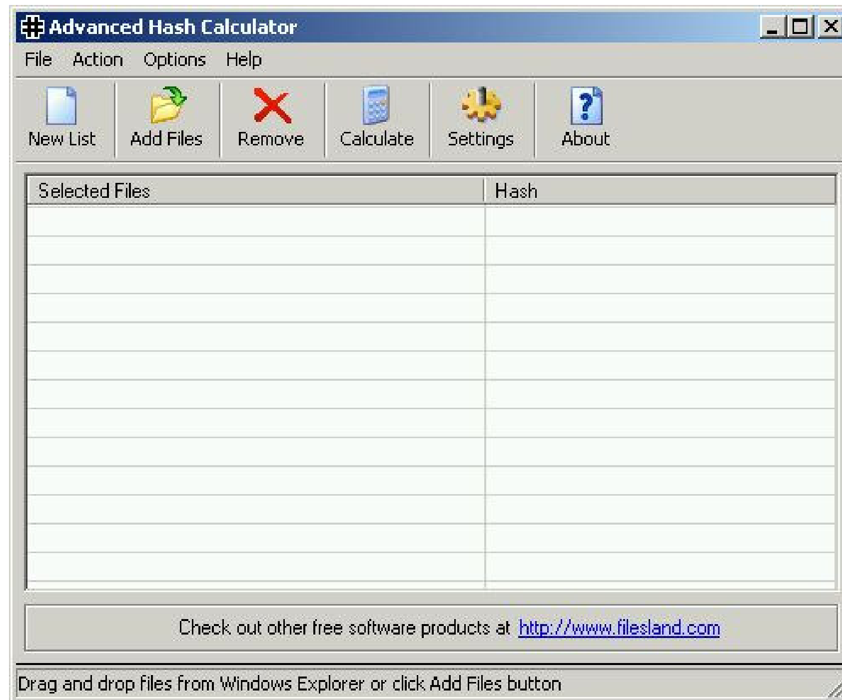

Figure 5-3 Advanced Hash Calculator

To test the design in the software environment the UART module is discarded and the program and message block is directly inputted to the design. The test and verification of the design in the software platform is done by behavioral and timing simulations. In order to perform behavioral and timing simulations, test bench is created for the top module. In the test bench necessary input signals for the design is provided and the outputs are compared with the expected results. ModelSim XE III 6.0a is used as a HDL based simulation and debug environment. ModelSim can be initialized from the Xilinx Project Navigator. When the designer launches ModelSim using Xilinx design environments, the wave window appears. It contains waveforms for all input and output signals of the top-level design module. The output waveforms are observed in this window and desired analyses can be carried out.

The test and verification of the design on the hardware is done after the FPGA is configured. The input message and the program are entered to the design via UART interface from a PC by means of a simple user interface. The

message padding operation is done by this user interface software. The snapshot of the user interface is shown below in Figure 5-4.

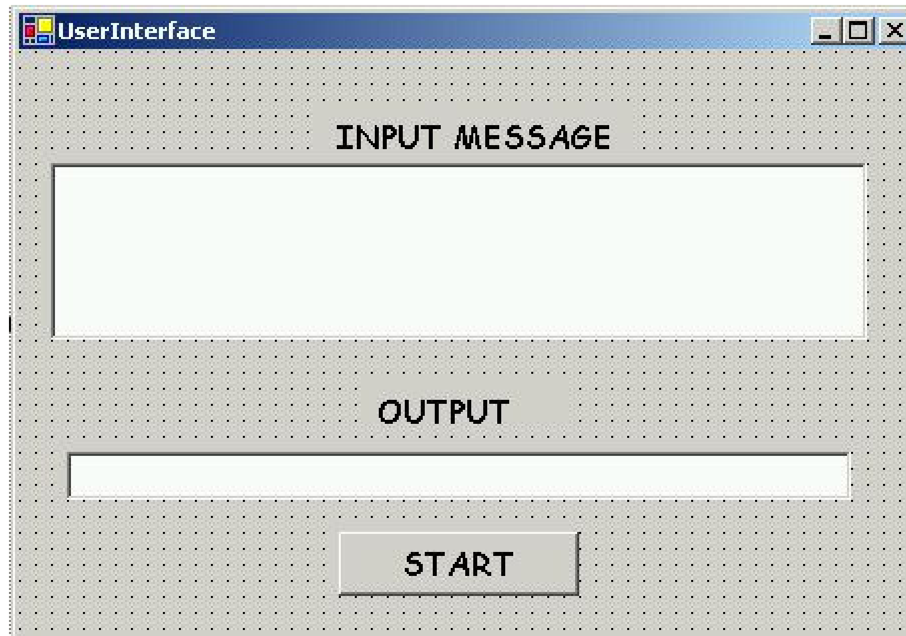

Figure 5-4 Hash Processor User Interface

5.3 TEST AND SIMULATION RESULTS

The designed hash processor is tested with different inputs. The program which is used for SHA-1 calculation is shown below in Table 5-2.

Table 5-2 SHA-1 Calculation Program

Instruction	Description
LDA 1E:Loop	Load accumulator A with the content of the memory location 1E (the content of the memory location is initially zero)
RRGF1	read the intermediate hash values of the SHA-256 algorithm from the register file.
SHA1	generate the necessary control signals fort he datapath to make

Instruction	Description
	one step SHA-256 calculation
SRGF1	store the intermediate hash values of the SHA-256 algorithm to the register file
LDA 1E	Load accumulator A with the content of the memory location 1E
ADD 1D	ADD the content of the memory location 1D to the value in accumulator A (the content of the memory location 1D is one)
STA 1E	Store the content of the accumulator A to the memory location 1E
LDA 1F	Load accumulator A with the content of the memory location 1F(the content of the memory location 1F is 80)
SUB 1E	Subtract the content of the memory location 1E from the value in the accumulator A
JMPZ 09	Jump to Halt if the result of the previous subtraction is zero (ie 80 round is completed)
JMPP 00	Jump to Loop is the result of the previous subtraction is positive (ie 80 round is not completed)
HALT:Halt	Terminate the execution

The program which is used for SHA-256 calculation is shown below in Table 5-3.

Table 5-3 SHA-256 Calculation Program

Instruction	Description
LDA 1E:Loop	Load accumulator A with the content of the memory location 1E (the content of the memory location is initially zero)
RRGF2	read the intermediate hash values of the SHA-256 algorithm from the register file.
SHA2	generate the necessary control signals fort he datapath to make one step SHA-256 calculation
SRGF2	store the intermediate hash values of the SHA-256 algorithm to the register file
LDA 1E	Load accumulator A with the content of the memory location 1E
ADD 1D	ADD the content of the memory location 1D to the value in accumulator A (the content of the memory location 1D is one)

Instruction	Description
STA 1E	Store the content of the accumulator A to the memory location 1E
LDA 1F	Load accumulator A with the content of the memory location 1F(the content of the memory location 1F is 64)
SUB 1E	Subtract the content of the memory location 1E from the value in the accumulator A
JMPZ 09	Jump to Halt if the result of the previous subtraction is zero (ie 64 round is completed)
JMPP 00	Jump to Loop is the result of the previous subtraction is positive (ie 64 round is not completed)
HALT:Halt	Terminate the execution

In the below Figure 5-5 timing simulation results of the SHA-256 calculation with the program in the Table 5-3 is shown. The input vector is the string “abc”.This test vector is determined by FIPS 180-2 [2], expected output for the first hash variable is: 506e3058. As seen in Figure 5-5 the design generates the correct value. The whole hash output is not given as output in timing simulations considering the IO restrictions of the selected device.

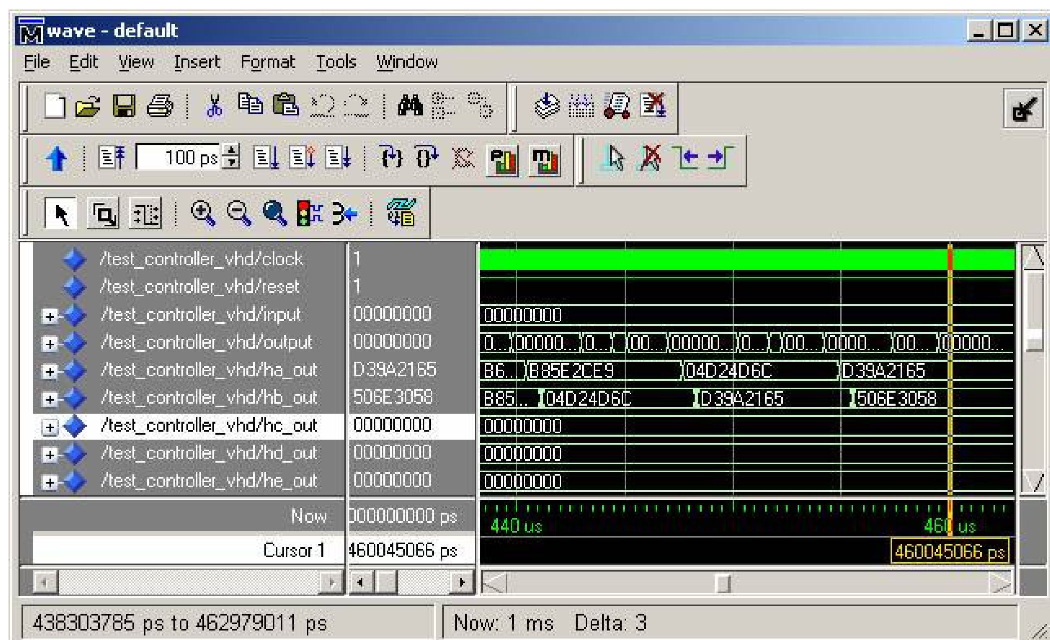

Figure 5-5 SHA-256 Calculation for Input “abc”

In the below Figure 5-6 timing simulation results of the SHA-1 calculation for the program given in Table 5-2 is shown. The input vector is the string “abc”.This test vector is determined by FIPS 180-2 [2], expected output for the first hash variable is: 42541B35.

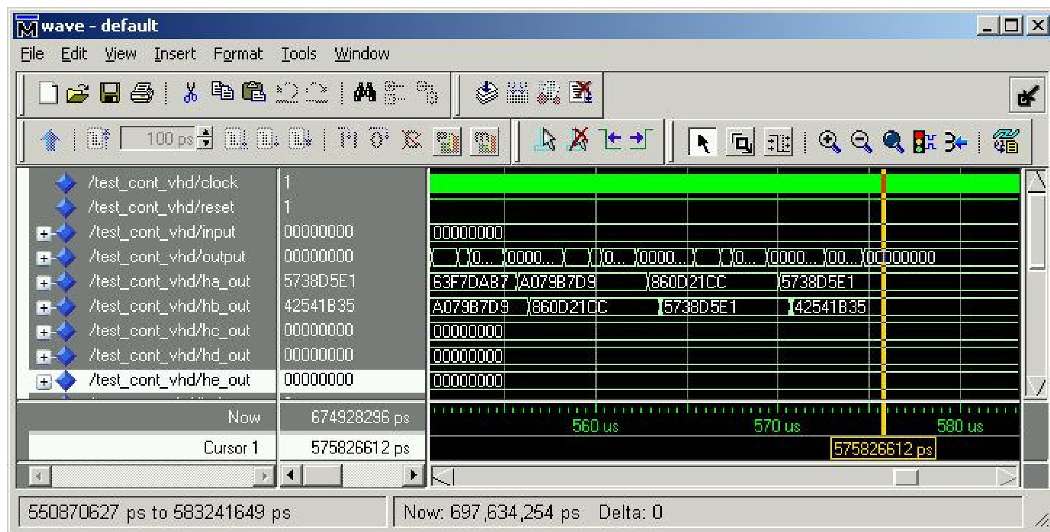

Figure 5-6 SHA-1 Calculation for Input “abc”

The hash processor is tested for random inputs too. In the below Figure 5-7, the result of the first chaining variable for input “tugba” for SHA-1 calculation is given. It is seen that the final value of the first chaining variable is “B3AF9A0B“. The first word of the final hash value is calculated as follows:

67452301 + B3AF9A0B = 1AF4BD0C

In order to verify the design the same string “tugba” is input to the AHC software. The software is configured for SHA-1 calculation and the output is observed. It is seen that both the AHC software and the proposed design produce the same outputs.

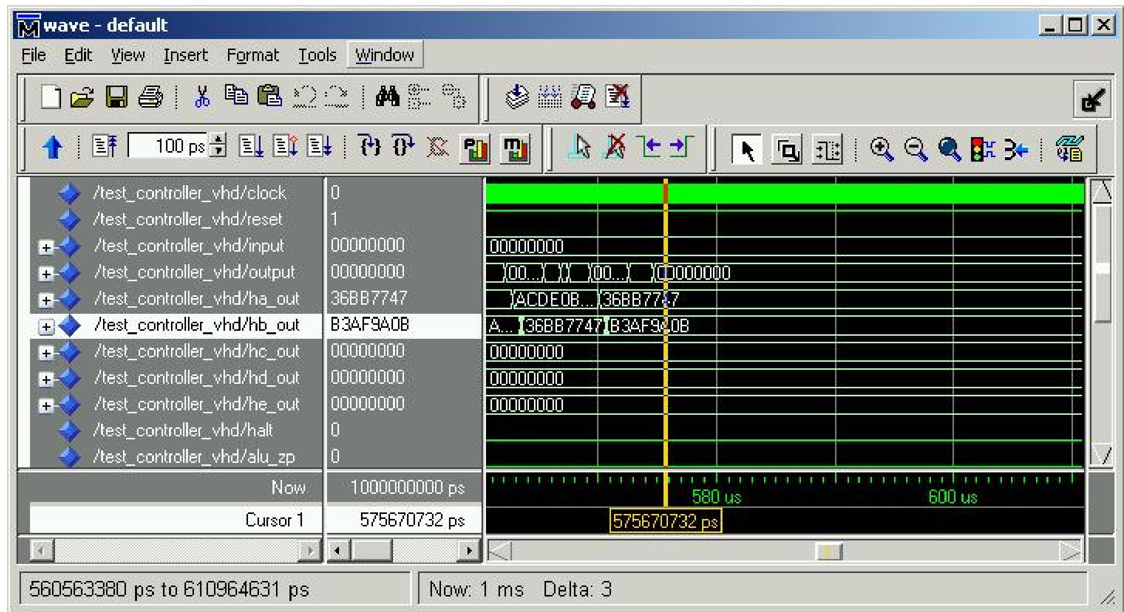

Figure 5-7 SHA-1 Calculation for Input "tugba"

Figure 5-8 SHA-1 Output of AHC for Input "tugba"

CHAPTER V

DISCUSSION AND CONCLUSION

In this thesis, a hash processor having the capability of performing SHA-1 and SHA-256 calculations is specified, analyzed and implemented using the hardware description language VHDL. The design is also verified on hardware by implementing the suggested structure on an FPGA.

Hash function implementations on hardware seem to be more popular as the developments in the communications area continue tremendously. Implementing hash functions on hardware is preferred since software implementations don't satisfy the speed, throughput and security requirements of the complex communication systems in use today. Hash function implementations are used in several fields of information security such as providing password authentication, verifying data integrity and generating digital signatures for both data origin authentication and verifying the content of the document. The hash processor proposed in this study can be used in these applications easily. The usage of the processor is flexible, since it has a serial communication interface that makes the communication with the external world possible.

The idea of implementing the hash functions SHA-1 and SHA-256 in a processor structure rose after a detailed research on present implementations of these hash functions in the market and in the literature. Moreover, the designed hash function processor has standard UART interface that makes communication

with the external units such as a personal computer possible. This provides a great flexibility since it enables remote control of the hash function processor. In the market and in the literature an implementation that enables serial communication with the device has not been found. This communication facility is proposed and added in this study. The proposed design is verified on software by timing simulations and on hardware by implementing the design on an FPGA. For the verification, test vectors announced by NIST are used and seen that the design generates the correct hash values. For random inputs which are not in the NIST publications "Advanced Hash Calculator (AHC)" software is used. When testing hash processor for random inputs, first, AHC software is verified by NIST's test vectors. Then the proposed design is verified by comparing outputs generated by the AHC software and the designed processor.

In order to define an architecture that implements hash functions, the computational properties of the hash functions are examined in details. The computational properties of the hash functions differ from each other by the parameters such as number of rounds, number of constants, message block sizes, word sizes and the complex logic and the arithmetic functions being used. In present implementations, hash functions are implemented as a combination of dedicated modules such as message padding unit, message scheduling unit, hash calculation unit and output generation unit. These implementations exhibit higher throughput.

However in this study a different design approach is followed. The computational properties of the hash functions are examined in details in order to define instructions that are specific to the hash functions and enable performing hash operations. Thus a dedicated instruction set for the processor under construction has been developed. Hash processor is a 32-bit processor with simple instruction set. The instruction set is composed of 14 instructions. The instructions are 2 clock cycle instructions. SHA-1 and SHA-256 calculations can be completed with 10 instructions. All the instructions are 32-bit words and kept in the program memory. This memory is addressed with 5-bit addresses. As an

addition, the instruction set of the hash processor can be extended easily to include other hash functions which have the same word and block sizes.

The proposed hash processor consists of the blocks control unit, program memory and the datapath which are the blocks that are present in all processors. The throughput of the proposed architecture is less than the present implementations however the proposed implementation has a serial communication interface which makes the design easy to use and consumes less area. Additionally, the architecture can be extended easily to include other hash functions since the general blocks will not be changed but some extra operations will be added to the ALU block for each hash function included.

The hash function processor described using the hardware description language VHDL is implemented on to the Xilinx Virtex4 4vsx35ff668-12 FPGA. The design consumes 1247 Configurable Logic Blocks (CLBs) on FPGA. This corresponds to the %16 of the FPGA CLBs. The modules of the hash function processor are designed in a synthesizable form in order to use the resources of the FPGA efficiently. For instance, the program memory and the message RAM are designed in an efficient manner such that they are implemented as Block RAMs (BRAM) on FPGA instead of consuming flip flops. Simulation results show that the throughput of the proposed architecture is 1,37 Mbps with a clock speed of 12.5 MHz. This is less than the designs present in the market however the proposed design provides standard UART communication interface can be controlled remotely and consumes less area on the FPGA.

Existing designs have some advantages such as high speed and high throughput however they can not be modified easily to include other hash functions. On the other hand, the ALU block which performs the arithmetic and the logic calculations can be optimized to improve the speed and the throughput of the design. As a future work, with the addition of some new instructions, the instruction set of the hash processor can be extended to include other hash functions such as SHA-224 and MD5.

REFERENCES

[1] Bruce Schneier, "*Applied Crptography*", John Wiley and Sons, Inc. Press, 1996.

[2] NIST, "Secure Hash Standard", FIPS PUB 180-1, May 1993.

[3] NIST, "with change notice Secure Hash Standard", FIPS PUB 180-2 August 2002.

[4] NIST, "Digital Signature Standard (DDS)", FIBS PUB 186 May 1994.

[5] "An Overview of Cryptographic Hash Functions and Their Uses", SANS Institute, 2003

[6] R. Rivest, "MD5 Message-Digest Algorithm", RFC 1321. MIT Laboratory for Computer Science and RSA Data Security Inc, 1992.

[7] Yong Kyu Kang, Dae Won Kim, Taek Won Kwon, Jun Rim Choi, "An Efficient Implementation of Hash Function Processor for IPSEC", In Proceedings of the IEEE Asia-Pacific Conference on ASIC, pp. 93-96, August. 2002

[8] M. McLoone, J. V. McCanny, "Efficient Single-Chip Implementation of SHA-384 & SHA-512", Proceedings of the IEEE International Conference on Field-Programmable Technology pp. 311-314, 2002.

[9] Tim Grembowski, Roar Lien, Kris Graj, Nghi Nguyen, Peter Bellows, Jaroslav Flidr, Tom Lehman, Brian Schott, "Comparative Analyses of the Hardware Implementations of Hash Functions SHA-1 and SHA-512", In: Proceedings. of the 5th International Information Security Conference, 2002

[10] N. Sklavos, G. Dimitroulakos, O. Koufopavlou, "An Ultra High Speed Architecture for VLSI Implementation of Hash Functions", Proceedings of the 10th IEEE International Conference on Electronics, Circuits and Systems (ICECS 2003) pp. 990-993 Vol.3, 2003.

[11] N. Sklavos and O. Koufopavlou, “On the Hardware Implementations of SHA-2 (256, 384, 512) Hash Functions”, Proceedings of the 2003 International Symposium on Circuits and Systems (ISCAS’ 2003) pp. V-153-V-156 Vol. 5, 2003

[12] Harris Michail, Athanasios P. Kakarountas, Odysseas Koufopavlou, Costas E. Goutis, “A Low Power and High Throughput Implementation of the SHA-1 Hash Function”, IEEE International Symposium on Circuits and Systems (ISCAS 2005), pp. 4086-4089 Vol.4., 2005

[13] T.S. Ganesh, M.T. Frederick, T.S.B. Sudarshan, A.K. Somani, “HashChip: A shared-resource multi-hash function processor architecture on FPGA”, INTEGRATION theVLSI journal 40 pp. 11-19, (2007).

[14] “Advanced Hash Calculator”, www.filesland.com, Version 2.33 Released November 21’st 2007.

[15] R. L. Rivest, “The MD4 Message Digest Algorithm,” RFC 1320, April 1992.

[16] J. Vandewalle, et al., “A European Call For Cryptographic Algorithms: RIPE; Race Integrity Primitives Evaluation”, EUROCRYPT ’89, LNCS 434, pp. 267-271, 1990.

[17] NIST, “SKIPJACK and KEA Algorithms Specifications Version 2.0”, May 1998.

[18] NIST, “Advanced Encryption Standard (AES),” FIPS PUB197, Nov. 2001.

[19] CAST, “SHA-1 Secure Hash Function Core Datasheet”, October 2007

[20] CAST, “SHA-256 Secure Hash Function Core Datasheet”, October 2007

[21] HDL Design House, “HCR_SHA1 Datasheet”, version 1.0, December 2002

[22] Helion Technology Limited, “SHA-1, SHA-256 and MD5 Hashing, Fast (Helion) Datasheet”, July 2005.

[23] Helion Technology Limited, “SHA-1, SHA-224, SHA-256 and MD5 Hashing, Tiny with HMAC (Helion) Datasheet”, July 2005.

[24] Aldec Inc, “ALDEC SHA IP Core Data Sheet”, version 1.0, April 2006.

[25] Ocean Logic Pty. Ltd, “OL_SHA256 SHA-256 Processor Datasheet”, Rev 0.9

[26] Ocean Logic Pty. Ltd, "OL_SHA SHA-1 Processor Datasheet", Rev 1.3

[27] Sci-worx, "High Speed SHA-1 HASH Engine Datasheet", Rev. 01.00.06

[28] Volnei A. Pedroni, "*Circuit Design with VHDL*", MIT Press Cambridge, Massachusetts, 2004.

[29] Clive Maxfield, "*The Design Warrior's Guide to FPGAs",* Elsevier Newnespress, 2004.

[30] Enoch O. Hwang, "*Digital Logic and Microprocessor Design With VHDL*", Team ELECTRONICS, 2004.

[31] Xilinx Inc., ML40x Evaluation Platform Users Guide, Feburary 2005.

[32] Xilinx Inc., Virtex-4 User Guide, October 2006.

[33] Xilinx Inc., Platform Cable USB Advance Product Specifications, June 2006.

APPENDICES

APPENDIX-A

SHA-1 AND SHA-256 CONSTANTS

The constants used by the SHA-1 and SHA-256 are given below in Table A-1 and Table A-2. The values in below tables are expressed in hex form.

Table A-1 SHA-1Constants

Constant	SHA-1 round number ‘t’
5a827999	$0 \leq t \leq 19$
6ed9eba1	$20 \leq t \leq 39$
8f1bbcdc	$40 \leq t \leq 59$
ca62c1d6	$60 \leq t \leq 79$

Table A-2 SHA-256 Constants

Constant	SHA-256 round number ‘t’
428a2f98	0
71374491	1
b5c0fbcf	2
e9b5dba5	3
3956c25b	4
59f111f1	5
923f82a4	6
ab1c5ed5	7
d807aa98	8
12835b01	9
243185be	10

Constant	SHA-256 round number ‘t’
550c7dc3	11
72be5d74	12
80deb1fe	13
9bdc06a7	14
c19bf174	15
e49b69c1	16
efbe4786	17
0fc19dc6	18
240ca1cc	19
2de92c6f	20
4a7484aa	21
5cb0a9dc	22
76f988da	23
983e5152	24
a831c66d	25
b00327c8	26
bf597fc7	27
c6e00bf3	28
d5a79147	29
06ca6351	30
14292967	31
27b70a85	32
2e1b2138	33
4d2c6dfc	34
53380d13	35
650a7354	36
766a0abb	37
81c2c92e	38
92722c85	39

Constant	SHA-256 round number ‘t’
a2bfe8a1	40
a81a664b	41
c24b8b70	42
c76c51a3	43
d192e819	44
d6990624	45
f40e3585	46
106aa070	47
19a4c116	48
1e376c08	49
2748774c	50
34b0bcb5	51
391c0cb3	52
4ed8aa4a	53
5b9cca4f	54
682e6ff3	55
748f82ee	56
78a5636f	57
84c87814	58
8cc70208	59
90befffa	60
a4506ceb	61
bef9a3f7	62
c67178f2	63

APPENDIX B

COMMERCIAL HASH IMPLEMENTATIONS

B.1 CAST SHA-1 SECURE HASH FUNCTION CORE

CAST SHA-1 Secure Hash Function Core consists of two main blocks; the SHA1 Engine Module and the Input Interface Module. The SHA1 Engine Module applies the SHA1 loops on a single 512-bit message block, while the Input Interface Module performs the message padding. The features of the core are as follows:

- Bit padding is provided.
- Supported Message lengths multiple of 8-bits.
- 82 processing cycles per message block.
- Fully stallable input and output interfaces, ideal for streaming applications.
- Optimized design for ASIC or FPGA implementations.
- Sophisticated self-checking Testbench (Verilog versions use Verilog 2001).

The functional block diagram of the core is given below in Figure B-1.

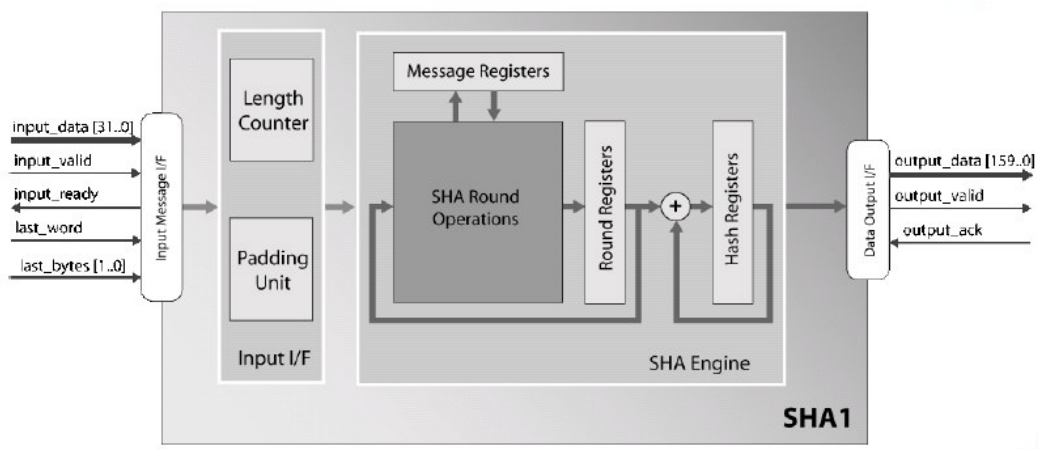

Figure B-1 CAST SHA-1 Secure Hash Function Core Block Diagram

B.2 CAST SHA-256 SECURE HASH FUNCTION CORE

CAST SHA-256 Secure Hash Function Core consists of two main blocks; the SHA256 Engine Module and the Input Interface Module. The SHA256 Engine Module applies the SHA256 loops on a single 512-bit message block, while the Input Interface Module performs the message padding. The features of the core are as follows:

- Bit padding is provided.
- Supported Message lengths multiple of 8-bits.
- 66 processing cycles per message block.
- Fully stallable input and output interfaces, ideal for streaming applications.
- Optimized design for ASIC or FPGA implementations.
- Sophisticated self-checking Testbench (Verilog versions use Verilog 2001).

The functional block diagram of the core is given below in Figure B-2.

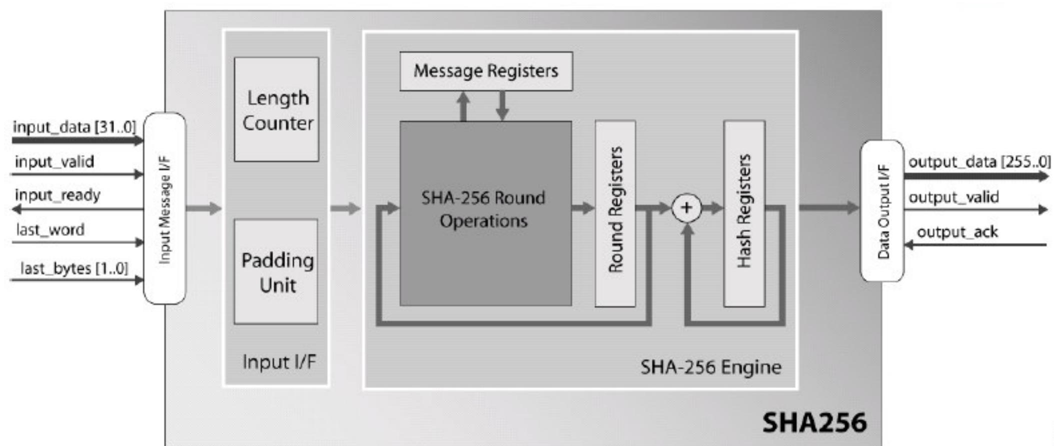

Figure B-2 CAST SHA-256 Secure Hash Function Core Block Diagram

B.3 HDL DESIGN HOUSE HCR_SHA1

The HCR_SHA1 is a high performance crypto core family that implements the NIST SHA-1 message-digest algorithm. The features of the core are as follows:

- The resolution of the input message is in bits
- All padding variants are supported in hardware
- Message block of 512 bits processed in 64 clock cycles
- Software configurable IP core through ten registers. Fixed 32 bits size of all architecture registers
- 640 Mb/s transfer rate for 100MHz OCP interface variant
- Input synchronous FIFO with concurrent read/write for input data stream
- Power down mode operation for low power applications
- Available in both Verilog and VHDL
- VITAL 2000 and SystemC behavioral models
- DFT support implemented
- SoC integration support

The functional block diagram of the core is shown below in Figure B-3.

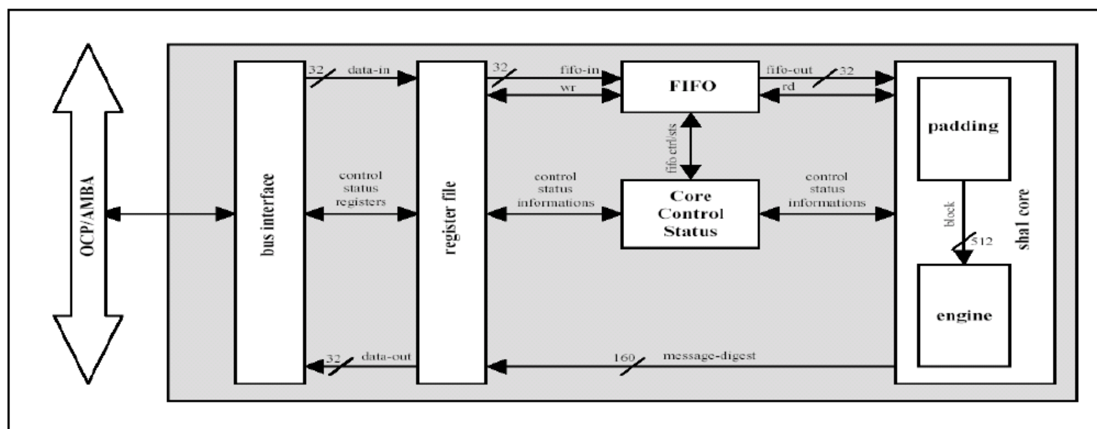

Figure B-3 HDL Design House HCR_SHA1 Core Block Diagram

B.4 HELION TECHNOLOGY LIMITED SHA-1, SHA-256 AND MD5 HASHING, FAST (HELION)

Hellion Technology Limited fast hashing core is capable of performing SHA-1, SHA-256 and MD5 hashing. The features of the core are as follows:

- Available in multiple versions
  - SHA-1 only
  - SHA-256 only
  - MD5 only
  - Dual-mode (selectable SHA-1 and SHA-256)
  - Dual-mode (selectable SHA-1 and MD5)
- Designed specifically for high throughput applications
- Performs automatic message length calculation and padding insertion
- Message is input as 32-bit words

Functional block diagram of the core is shown below in Figure B-4.

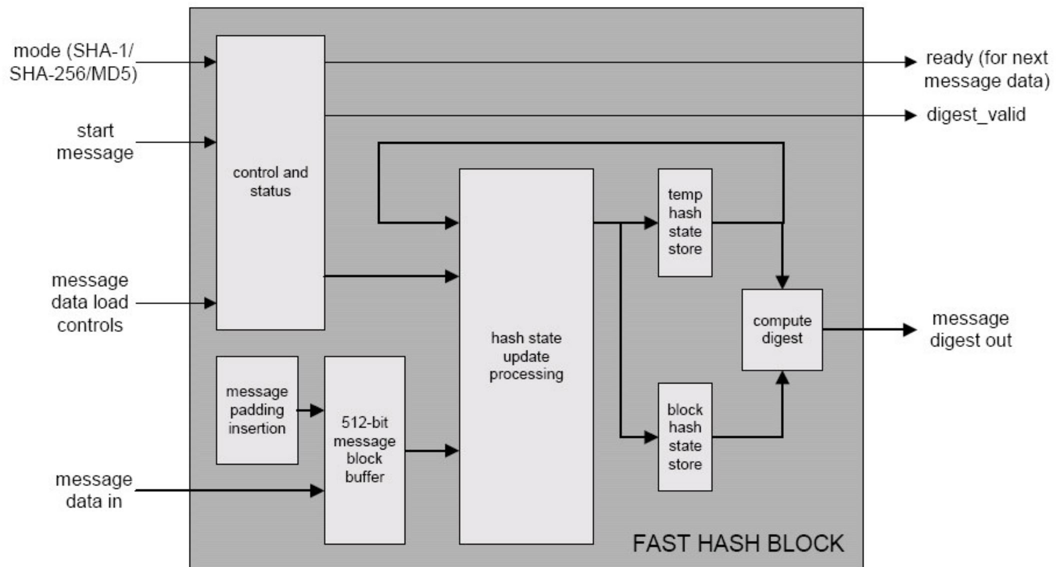

Figure B-4 Hellion Fast Hashing Core Block Diagram

B.5 HELION TECHNOLOGY LIMITED SHA-1, SHA-224, SHA-256 AND MD5 HASHING, TINY WITH HMAC

Hellion Technology Limited tiny hashing core is capable of performing SHA-1, SHA-224, SHA-256 and MD5 hashing. The features of the core are as follows:

- Supports MD5, SHA-1, SHA-224 and SHA-256 hash algorithms
- Supports Internet Standard HMAC for all four hash algorithms
- Supports state unload/reload to optimise handling of fragmented message streams
- Choice of 8, 16 or 32-bit data interface widths
- Highly flexible, low resource hashing solution for lower data rate applications
- Highly optimized for use in Xilinx FPGA technologies

Functional block diagram of the core is shown below in Figure B-5.

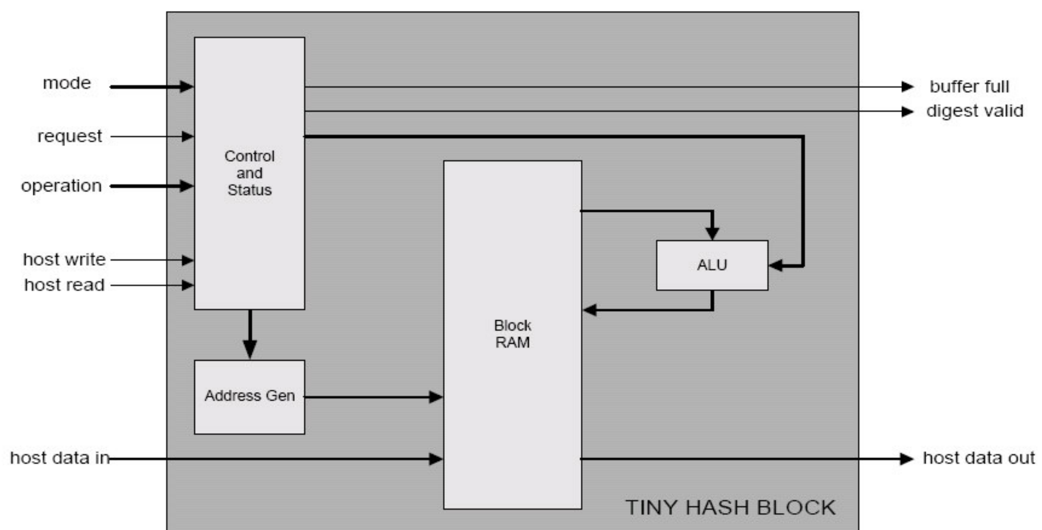

Figure B-5 Hellion Tiny Hashing Core Block Diagram

B.6 ALDEC INC ALDEC SHA IP CORE

ALDEC SHA IP CORE has the following features:

- Byte oriented hash calculation
- Hash value of 512-bit message is calculated in 81 clock cycles
- No dead clock cycles
- Simple interface and timing
- Fully synchronous design

Functional block diagram of the core is shown below in Figure B-6.

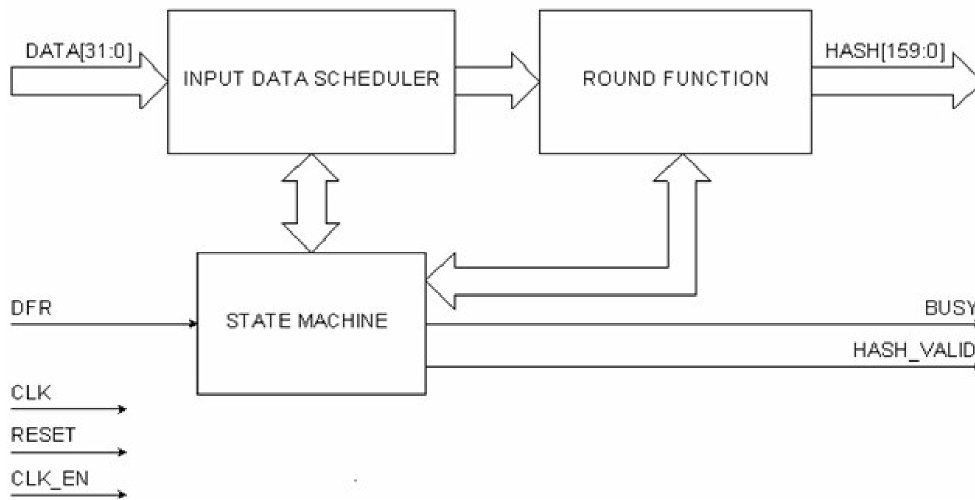

Figure B-6 ALDEC SHA IP Core Block Diagram

B.7 OCEAN LOGIC PTY. LTD OL_SHA256 SHA-256 PROCESSOR

The features of the core are as follows:

- FIPS 180-2 compliant.
- Suitable for data authentication applications.
- Fully synchronous design.

- Available as fully functional and synthesizable VHDL or Verilog soft-core.
- FPGA netlist available for various devices.

Functional block diagram of the OL_SHA256 SHA-256 Processor is shown below in Figure B-7.

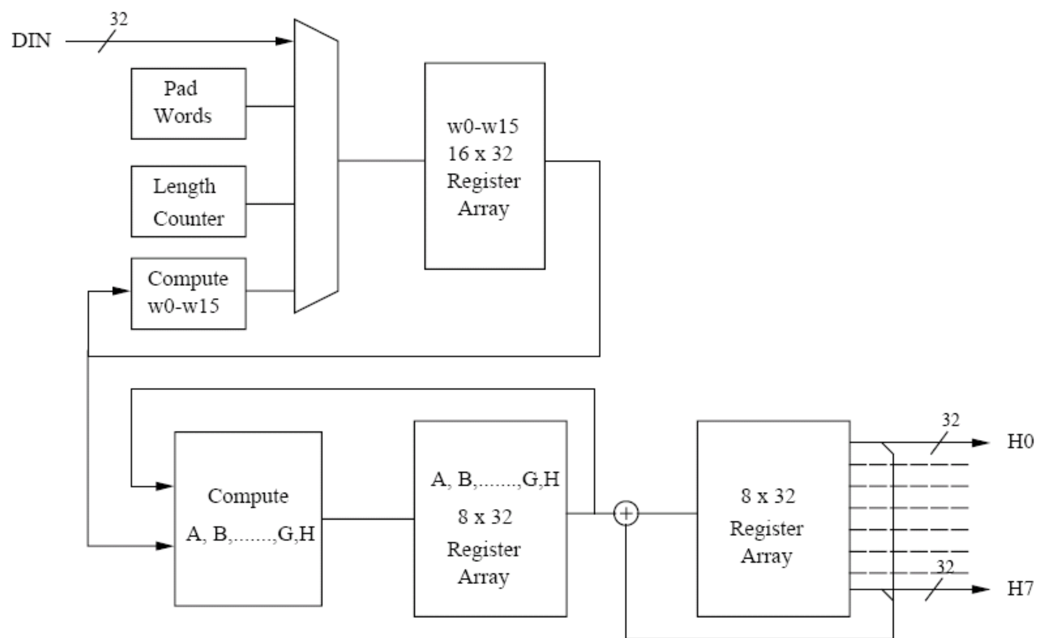

Figure B-7 Ocean Logic Pty. Ltd SHA-256 Processor Block Diagram

B.8 OCEAN LOGIC PTY. LTD OL_SHA SHA-1 PROCESSOR

The features of the OL_SHA SHA-1 Processor are as follows:

- Suitable for data authentication applications.
- Fully synchronous design.
- Available as fully functional and synthesizable VHDL or Verilog soft-core.
- Xilinx and Altera netlist available for various devices.

The functional block diagram of the core is shown below in Figure B-8.

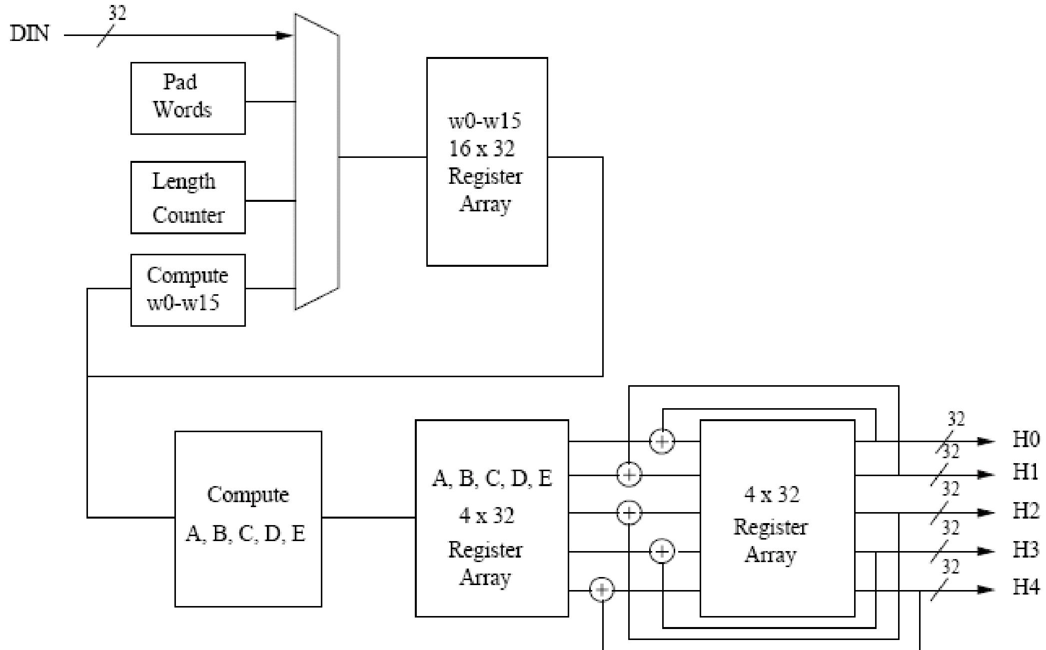

Figure B-8 OL_SHA SHA-1 Processor Core Block Diagram

B.9 SCI-WORX HIGH SPEED SHA-1 HASH ENGINE

The features of the Sci-worx High Speed SHA-1 HASH Engine are as follows:

- FIPS-180-1 compliant
- Fully synchronous single phase design
- Up to 140 MHz system clock (0.18 TSMC)
- Data rate 6.24 Mbit/s per MHz (830 Mbit/s@133 MHz)
- Source code available in VHDL and Verilog

Functional block diagram of the core is shown below in Figure B-9.

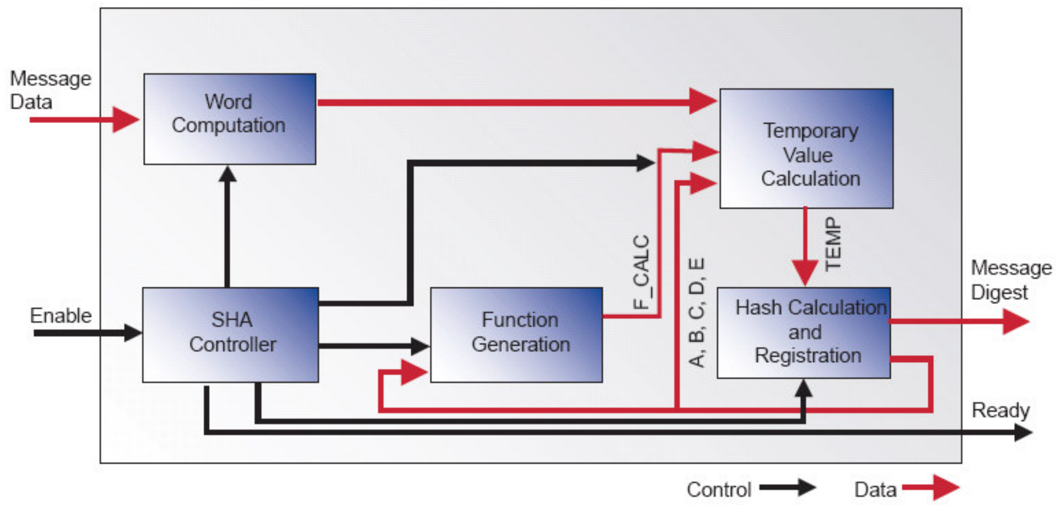

Figure B-9 Sci-worx High Speed SHA-1 HASH Engine Block Diagram

APPENDIX C

STRUCTURE OF CD-ROM DIRECTORY

The source codes and executable files of the simulations performed in this study are given in the CD attached at the back cover of this thesis. The contents of the CD are given below in Table C-1.

Table C-1 Structure of CD-ROM Directory

\SRC	VHDL Source Files
\TestBench	VHDL Test Bench Files
\SIM	VHDL Simulation Results