VHDL IMPLEMENTATION OF A SECURITY CO-PROCESSOR

by

Scott Wakelin
B.A.Sc., Engineering Science
Simon Fraser University

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF

MASTER OF APPLIED SCIENCE

in the School
of
Engineering Science

© Scott Wakelin 2005

SIMON FRASER UNIVERSITY

Summer 2005

All rights reserved. This work may not be reproduced in whole or in part, by photocopy or other means, without permission of the author.

APPROVAL

Name: Scott Wakelin

Degree: Master of Applied Science

Title of Thesis: VHDL Implementation of a Security Co-Processor

Examining Committee:

Chair: **Dr. Glenn Chapman**
Professor of the School of Engineering Science

____

Dr. Rick Hobson
Senior Supervisor
Professor of the School of Engineering Science

____

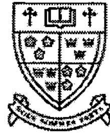
Dr. Ljiljana Trajkovic
Supervisor
Professor of the School of Engineering Science

____

Dr. Marek Syrzycki
Internal Examiner
Professor of the School of Engineering Science

Date Defended: ______________________________

SIMON FRASER UNIVERSITY



PARTIAL COPYRIGHT LICENCE

The author, whose copyright is declared on the title page of this work, has granted to Simon Fraser University the right to lend this thesis, project or extended essay to users of the Simon Fraser University Library, and to make partial or single copies only for such users or in response to a request from the library of any other university, or other educational institution, on its own behalf or for one of its users.

The author has further granted permission to Simon Fraser University to keep or make a digital copy for use in its circulating collection.

The author has further agreed that permission for multiple copying of this work for scholarly purposes may be granted by either the author or the Dean of Graduate Studies.

It is understood that copying or publication of this work for financial gain shall not be allowed without the author's written permission.

Permission for public performance, or limited permission for private scholarly use, of any multimedia materials forming part of this work, may have been granted by the author. This information may be found on the separately catalogued multimedia material and in the signed Partial Copyright Licence.

The original Partial Copyright Licence attesting to these terms, and signed by this author, may be found in the original bound copy of this work, retained in the Simon Fraser University Archive.

W. A. C. Bennett Library
Simon Fraser University
Burnaby, BC, Canada

ABSTRACT

Tradeoffs of speed vs. area that are inherent in the design of a security co-processor are explored. Encryption, decryption, and key generation engines for AES in Cipher Block Chaining and Electronic Code Book modes were developed using VHDL. Two designs are discussed.

The “space-optimised” design required 1454 FPGA CLB slices for the Cipher implementation (4016 for the complete design) and produced a round delay of ~ 16.75 ns. The throughput in CBC mode was 636.82 Mbps (depending on the FPGA utilized), which is greater than various published prior works.

The Multi-Session Pipelined approach followed a novel architecture that required 13675 CLB slices total and produced a round delay of ~ 20 ns. The Multi-Session Pipelined AES design can obtain an aggregate throughput of ~ 6.40 Gbps and is capable of operating in CBC mode. The 10x speedup over the “space-optimised” design required 3.4x the total number of FPGA CLB slices.

DEDICATION

To my wife Stacey and son Evan who provide endless love, support, and inspiration. And to our new (yet to be born) baby, who perhaps provided the greatest inspiration of all. Finally, to my Mom and Dad, for without your loving guidance and encouragement throughout my life, this would not have been possible.

ACKNOWLEDGEMENTS

I would like to give a special thanks to Dr. Hobson whose direction and advice was a driving force behind this work. Your knowledge and expertise were irreplaceable in the completion of the Thesis and I hope we can continue to work on our ideas together in the future.

I would also like thank Dr. Syrzycki for his guidance throughout my entire academic career (undergraduate and graduate) at Simon Fraser University. I was honoured to be a student in several of your classes, and look forward to the opportunity to learn from you again.

I would also like to thank Dr. Trajkovic and Dr. Chapman. It has been a pleasure to learn from you during my time at Simon Fraser University. I have met few people as dedicated as you.

I also wish to acknowledge the kind and generous support of the Science Council of British Columbia.

Finally, I wish to thank PMC-Sierra for providing all necessary support and assistance in the completion of my degree. PMC-Sierra is a great company whose strengths are its people, products and leadership. Thank you all for being such great friends.

TABLE OF CONTENTS

Approval ii

Abstract iii

Dedication iv

Acknowledgements v

Table of Contents vi

List of Figures viii

List of Tables x

List of Abbreviations and Acronyms xi

Chapter 1 INTRODUCTION 1

Chapter 2 VIRTUAL PRIVATE NETWORKS 3

Chapter 3 INTERNET PROTOCOL SECURITY (IPSEC) 8

3.1 IPSec Protocols 8
3.1.1 IPSec Protocol Modes 9
3.2 IPSec Security Associations and Policy 12
3.3 IPSec Processing 12
3.3.1 Inbound Processing 13
3.3.2 Outbound Processing 14

Chapter 4 ENCRYPTION ALGORITHMS 15

4.1 Modes of Operation 16
4.1.1 Electronic Code Book (ECB) Mode 17
4.1.2 Cipher Block Chaining Mode 18
4.2 The Data Encryption Standard (DES) 20
4.2.1 Triple-DES (3-DES) 23
4.3 The Advanced Encryption Standard 24
4.3.1 AES Cipher 25
4.3.2 AES Inverse Cipher 29
4.3.3 AES Key Expansion 31

Chapter 5 DESIGN AND IMPLEMENTATION OF AES 34

5.1 Architectural Options 34
5.1.1 Pipelining 35
5.1.2 Sub-Pipelining 36
5.1.3 Loop Unrolling 36
5.1.4 Multi-Session Pipelining 40

5.2 Algorithmic Options 42
5.3 Implementation Options 47

Chapter 6 DETAILED DESIGN OF AES 49

6.1 Space Optimised AES Cipher 49
6.1.1 Input FIFO Sub-module 52
6.1.2 Output FIFO Sub-Module 53
6.1.3 Control SM Sub-module 54
6.1.4 AES Cipher Sub-module 57
6.2 Multi-Session Pipelined AES Cipher 60
6.2.1 AES Cipher Sub-module 62
6.3 Inverse Cipher Design 64
6.4 Key Expansion Design 65

Chapter 7 AES DESIGN VERIFICATION 66

7.1 Space-Optimised AES Testbench Design 66
7.1.1 Input Interface 67
7.1.2 Output Interface 68
7.1.3 Key Interface 69
7.2 Multi-Session Pipelined AES Testbench Design 70
7.3 AES Cipher Module Verification 70

Chapter 8 AES RESULTS 74

8.1 Space Optimised AES Design Results 74
8.2 Multi-Session Pipelined AES Design Results 77
8.3 FPGA, ASIC and Full Custom Design Results 79
8.4 Summary of Results 81

Chapter 9 REALIZATION OF A SECURITY CO-PROCESSOR 82

Chapter 10 CONCLUSION 84

APPENDIX A – SIMULATION RESULTS 86

APPENDIX B – RTL CODE 95

REFERENCES 129

LIST OF FIGURES

Figure 1 Private Leased Line LAN Interconnect (logical view) 3
Figure 2 Frame Relay Network Alternative to Private Leased Lines (simplified view) 4
Figure 3 Classification of VPN Types 6
Figure 4 ESP Mode Packet Format 9
Figure 5 ESP Packet in Tunnel and Transport Mode 10
Figure 6 Tunnel Mode Example 11
Figure 7 Transport Mode Example 11
Figure 8 Electronic Code Book mode 17
Figure 9 Cipher Block Chaining mode, Encryption 19
Figure 10 Cipher Block Chaining mode, Decryption 20
Figure 11 DES Structure 21
Figure 12 DES Round Function 22
Figure 13 Triple-DES 24
Figure 14 The AES State 25
Figure 15 AES Cipher 26
Figure 16 AES Round Function 26
Figure 17 ShiftRows 28
Figure 18 AES Inverse Cipher 29
Figure 19 AES Inverse Cipher Round Function 30
Figure 20 InvShiftRows 30
Figure 21 AES Key Expansion 33
Figure 22 Pipeline architecture with K = 1 35
Figure 23 Pipelined vs. Loop Unrolled Architectures 37
Figure 24 Multi-Session Pipeline System Diagram 41
Figure 25 AES Cipher Round Algorithm 43
Figure 26 Unbalanced MixColumns implementation 45
Figure 27 AES Cipher Module Block Diagram 50
Figure 28 Control SM State Diagram 55

Figure 29 AES Cipher Sub-module Block Diagram..57
Figure 30 AES Cipher Module Using T-Box Approach 60
Figure 31 Multi-Session Pipelined AES Cipher Module 61
Figure 32 AES Inverse Cipher in CBC Mode 64
Figure 33 Testbench Connections..67
Figure 34 Functional Timing Diagram of the Input Interface 68
Figure 35 Output Interface Functional Timing..69
Figure 36 Key Interface Functional Timing 70
Figure 37 AES Cipher Encryption..73
Figure 38 Block Diagram of the Complete AES Processor 83
Figure 39 Simulation Result of the Space Optimised Cipher (Full View)........................87
Figure 40 Simulation Result of the Space Optimised Cipher (ECB Section)..................88
Figure 41 Simulation Result of the Space Optimised Cipher (CBC Section)..................89
Figure 42 Simulation Result of the Space Optimised Cipher (CBC Section, Part 2) 90
Figure 43 Simulation Result of the Multi-Session Pipelined Cipher (Full View) 91
Figure 44 Simulation Result of the Multi-Session Pipelined Cipher (Inputs)...................92
Figure 45 Simulation Result of the Multi-Session Pipelined Cipher (ECB and CBC outputs)...93
Figure 46 Simulation Result of the Multi-Session Pipelined Cipher (Last CBC output)...94

LIST OF TABLES

Table 1 Round Constant (RCON) Values for Key Expansion 32
Table 2 Speedup achieved by using loop-unrolling..39
Table 3 Key Physical Implementation Characteristics 47
Table 4 Pin Description of Space Optimised AES Cipher Module..................................51
Table 5 Pin Description of the Multi-Session Pipeline AES Cipher Module....................63
Table 6 Test Vectors used in the verification of AES 71
Table 7 Space Optimised AES Design Summary 75
Table 8 Performance Characteristics of the Space Optimised AES Design...................76
Table 9 Performance Characteristics with Same FPGA 76
Table 10 Multi-Session Pipelined AES Design Summary 78
Table 11 Performance Characteristics of the Multi-Session Pipelined AES Design 78
Table 12 Comparison of ASIC Speed and Size Requirements......................................80

LIST OF ABBREVIATIONS AND ACRONYMS

Acronym	Definition
2547	A type of MPLS-based VPN, defined in RFC-2547.
3-DES	Triple DES Encryption
AES	Advanced Encryption Standard, defined in Federal Information Processing Standards Publication 197
AH	Authenticating Header, defined in RFC-2402
ASIC	Application Specific Integrated Circuit
ATM	Asynchronous Transfer Mode
BDD	Binary Decision Diagram
CBC	Cipher Block Chaining
CLB Slice	Combinatorial Logic Block. A Xilinx specific term that refers to the reconfigurable units within each FPGA.
CTR	Counter Mode
DES	Data Encryption Standard, defined in Federal Information Processing Standards Publication 46-2
ECB	Electronic Code Book
ESP	Encapsulating Security Payload, defined in RFC-2406
FIFO	First-in, First-out
FPGA	Field Programmable Gate Array
FR	Frame Relay
GRE	Generic Routing Encapsulation
IKE	Internet Key Exchange
IP	Internet Protocol

IPSec	Internet Protocol Security
IV	Initialisation Vector
L2TP	Layer 2 Tunneling Protocol
L2VPN	Layer 2 Virtual Private Network
L3VPN	Layer 3 Virtual Private Network
LUT	Look-up Table
MPLS	Multi-Protocol Label Switching
PE	Provider Edge
PPP	Point to Point Protocol
RCON	Round Constant
ROM	Read-Only Memory
SA	Security Association
SADB	Security Association Data Base
SONET	Synchronous Optical Network
SPD	Security Policy Data Base
SPI	Security Parameter Index
TCP	Transmission Control Protocol
VHDL	Very High Speed Integrated Circuits Hardware Description Language
VPLS	Virtual Private LAN Service
VPN	Virtual Private Network
VPWS	Virtual Private Wire Service
XOR	Exclusive-OR

CHAPTER 1 INTRODUCTION

Over the past two decades, the Internet has evolved from its research-oriented roots to the ubiquitous network we know today that is accessed daily by hundreds of millions of people in all corners of the globe. We are all familiar with the most popular uses of the Internet, from email to web surfing.

Increasingly, the Internet is used as a medium for conducting business, whether it be E-commerce, or online banking. In addition, businesses are using the Internet as a means to connect often times geographically dispersed sites together, forming what is known as a Virtual Private Network (VPN). Finally, these same businesses require scalable and cost-effective solutions that enable their travelling workforce to access the company Intranet. A common element of all these new applications is the need for enhanced security.

A suite of protocols, collectively referred to as IPSec, was developed out of the need to secure the Internet Protocol (IP). The Internet Protocol suffers from a number of shortcomings, including the ease with which its header could be forged and payload snooped or altered. IPSec uses two protocols: the Encapsulating Security Payload (ESP) and Authenticating Header (AH) to address these shortcomings. AH provides data integrity, data origin authentication, and anti-replay protection, while ESP offers all the services provided by AH, and adds confidentiality services [1].

Encryption algorithms, such as the Data Encryption Standard (DES), 3-DES, or the Advanced Encryption Standard (AES) are used to provide the confidentiality

services. DES, 3-DES, and AES along with their public key counterparts such as RSA are computationally intensive algorithms that typically are implemented in software for low data rate applications, and FPGAs or ASICs for high data rate applications.

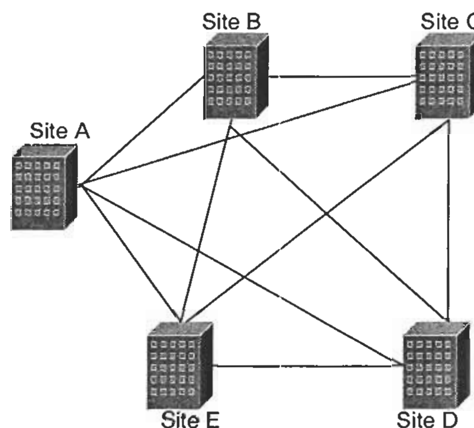
The goal of this thesis is to understand the issues in the design and implementation of a scalable and efficient security co-processor capable of supporting encryption and decryption at OC-12 data rates (622 Mbps). It is not the goal of this thesis to create the fastest AES implementation, but to provide a design that works in both CBC and ECB mode that meets the stated performance objective while making appropriate throughput/area trade-offs. The design is implemented in VHDL, and targeted for Xilinx FPGAs using Xilinx Foundation Series software. While the topic of this thesis is the design of a security co-processor, the scope of the VHDL implementation is limited to modules supporting the AES encryption algorithm.

The thesis is organized as follows: Virtual Private Networks are introduced and discussed in Chapter 2. IPSec, with particular focus on modes of operation is introduced in Chapter 3. Chapter 4 discusses both public-key and symmetric key encryption algorithms, with particular emphasis on DES and AES and the various modes of operation. Architectural and algorithmic design considerations are presented in Chapter 5. Detailed design and architectural descriptions of the various modules are presented in Chapter 6. Chapter 7 introduces the verification strategy of the design. Chapter 8 discusses the testing results for the modules, including a comparison with prior works. The integration of the modules into a security co-processor is presented in Chapter 9. Simulation results are contained in Appendix A, while the VHDL code developed for this Thesis is presented in Appendix B.

CHAPTER 2 VIRTUAL PRIVATE NETWORKS

As the Internet and corporate enterprise networks have evolved, businesses have sought the productivity and efficiency gains made possible by connecting their own sometimes geographically dispersed sites together, to form what is known as an Intranet. A corporate Intranet allows users at different sites within the same company to share information and collaborate in real time. Such flexibility, however, does not come without a cost. Traditional means for creating a corporate Intranet often meant purchasing and deploying costly private leased lines, which are dedicated, always on connections that typically run at T1 rates (1.544 Mbps) and above. Figure 1 shows an interconnection of 5 corporate sites in a full mesh of private leased lines. Note that Figure 1 only shows a logical view. Typically these leased lines are multiplexed with other links in access and metro SONET rings. Both are outside the scope of this paper.

Figure 1 Private Leased Line LAN Interconnect (logical view)



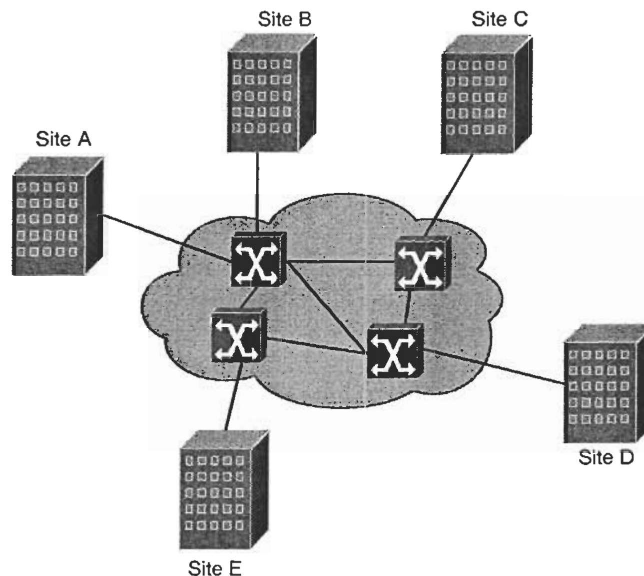
Private leased lines, though appropriate in some circumstances, have a number of important disadvantages most notably [2]:

- Cost: Both in terms of deployment, and operating

- Lack of scalability: Once the private line is in the ground, the bandwidth is fixed. In addition, as new sites are connected to the Intranet, new leased lines must be deployed to some or all other existing sites.

To overcome the obvious scalability hurdles inherent in a private leased line network, many businesses used Frame Relay (FR) or Asynchronous Transfer Mode (ATM) technology to connect their various sites together in what could be considered the first Layer-2 VPN. These networks solved the interconnection problem by allowing multiple virtual circuits to be multiplexed on the same physical link (and port). The service provider network was then responsible for ensuring virtual circuits were in place to create a hub and spoke topology that required fewer physical links. Figure 2 shows a simplified Frame Relay network alternative to the private leased line approach shown in Figure 2.

Figure 2 Frame Relay Network Alternative to Private Leased Lines (simplified view)



Such a deployment has significant disadvantages, most notably that it does not leverage the ubiquity of the growing IP-based Service Provider networks. In this case, the Service

Provider must maintain a FR based network in addition to its IP backbone. Furthermore, although the scalability of the network in Figure 2 is better than that of Figure 1, it does not offer the scalability inherent in an IP-based network.

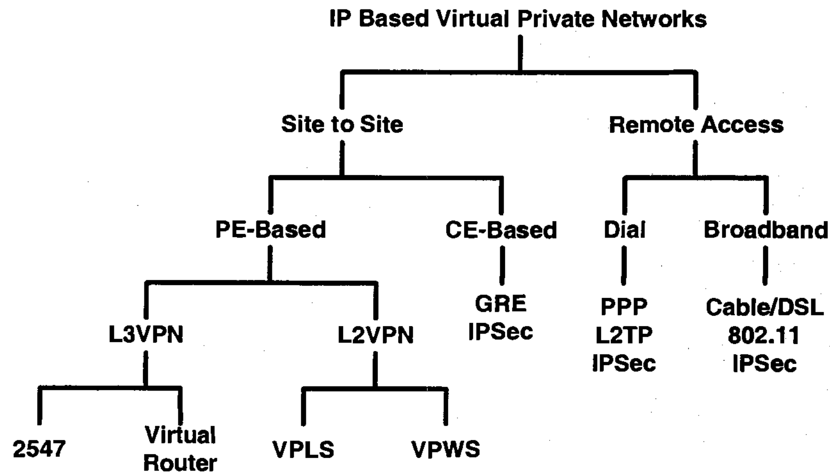
Another disadvantage of the described interconnect strategies is that it is cumbersome, if not impossible, to enable another emerging interconnect strategy, the Extranet. An Extranet is a business to business model that, for example, allows a supplier to access a companies inventory database to determine when additional shipments should be made based on demand and supply levels [2]. It would be economically unfeasible to install private leased lines to every one of a companies suppliers or customers.

Another evolution in the business and networking environments is the need for individual users, such as telecommuters, to connect to their corporate networks. Traditional approaches used slow and often costly (particularly if long distance charges were required) dial-up access.

With all this in mind, network equipment manufacturers and service providers began searching for solutions that allow scalable site-to-site, business-to-business, and user-to-site network access that leveraged the ubiquity of the Internet. The result was the development of the IP-based Virtual Private Network.

A Virtual Private Network (VPN) can be defined as a communication method that utilizes a segmentation of the existing shared network infrastructure to emulate a private network [2].

Figure 3 Classification of VPN Types



As shown in Figure 3, there are two general types of IP-Based VPNs: Site-to-Site, and Remote Access. The main differences between the two is the number of tunnels required to enable the VPN connectivity and the number of users of each individual tunnel [3].

Another common VPN classification scheme is based on whether the VPN is trusted or secure. A trusted VPN is one in which traffic belonging to the VPN stays within the confines of the VPN and is not mixed with general Internet traffic. MPLS and Frame Relay based VPNs are typical examples of a trusted VPN [4].

A secure VPN has some combination of encryption and/or authentication is applied to the traffic belonging to the VPN [3]. IPSec is a suite of security protocols that uses encryption and/or authentication facilities to protect traffic [1]. Secure, IPSec based VPNs are typically used for user-to-site and site-to-site connectivity.

Since IPSec implies the use of computationally intensive operations such as encryption and/or authentication, network devices implementing IPSec must have sufficient processing power to handle not only the IPSec functionality, but their normal routing and

forwarding roles as well. This often leads to the necessity to have a dedicated security co-processor. The focus of this Thesis is the design of AES modules, a key component of a security processor.

CHAPTER 3 INTERNET PROTOCOL SECURITY (IPSEC)

Today's Internet spans hundreds of millions of users and endpoints, and likely millions of content and service providers not all of whom can be trusted. Packets transmitted using the Internet Protocol are open to a wide range of rogue behaviour including: snooping, forging, modification, and replay. The IPSec protocol suite is an extension of IP designed to protect the data and authenticate the identity of those involved in the communication.

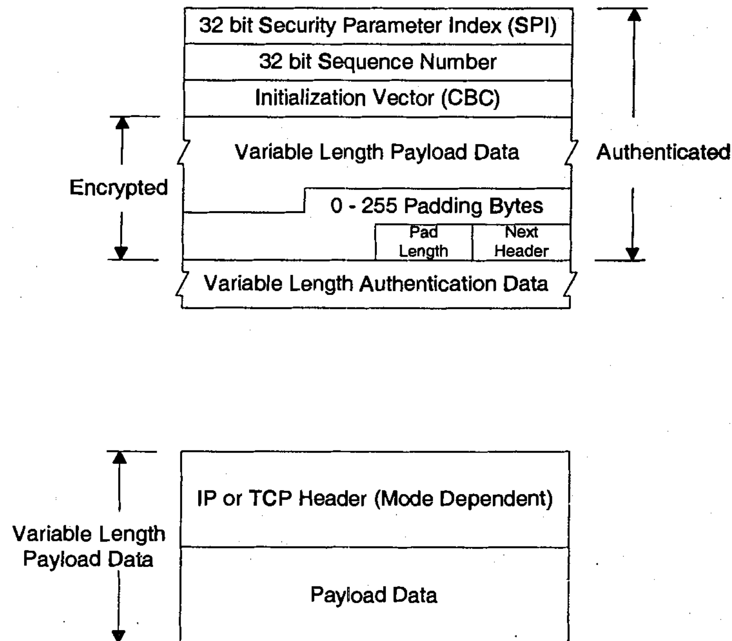
3.1 IPSec Protocols

IPSec defines two main protocols for securing IP traffic: AH and ESP. Authentication Header (AH), defined in RFC 2402 [8], provides data integrity, origin authentication, and anti-replay protection. Encryption services are not provided by AH, therefore, AH will not be discussed further in this Thesis.

Encapsulating Security Payload (ESP), defined in RFC 2406 [9], adds confidentiality (encryption) services to those provided by AH. Figure 4 shows the format of an ESP protocol packet [1]. The Security Parameter Index along with the packets source/destination address, and IPSec protocol value is used to identify the Security Association (SA) for a given packet. The SA dictates how security services are to be applied to a packet, including the cryptographic algorithms and associated keys [1], [6]. The sequence number is used to provide anti-replay protection. The variable length payload data contains the IP/TCP headers as well as the user data (if any) being transmitted. Padding is added to maintain alignment. Finally, an authentication word is added to provide data integrity verification. Note that the entire packet (other than the

authentication data) is authenticated. Encryption services are applied to the payload data, pad, pad length, and next header fields only [1], [6].

Figure 4 ESP Mode Packet Format



The variable length payload data contains an Initialization Vector (IV) when the encryption services dictate Cipher Block Chaining (CBC) mode should be used [9]. CBC mode will be discussed in further detail in section 4.1.2. The Initialization Vector may be any random data. Note that the IV is NOT encrypted [1], [6].

3.1.1 IPSec Protocol Modes

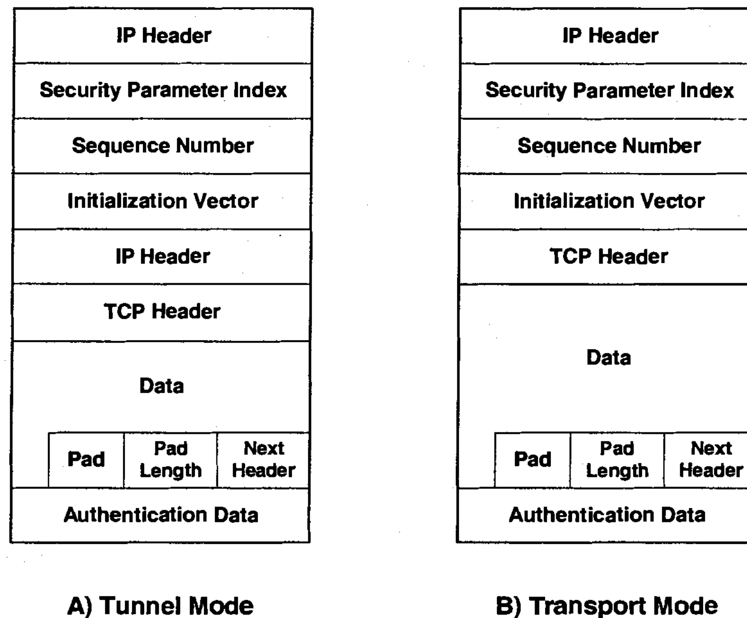
The IPSec protocols may operate in one of two modes: Tunnel or Transport [1], [6]. In tunnel mode, the entire IP packet is protected by ESP or AH and a second IP header is added on the outside. In this way, the protected IP packet may be tunnelled through a network without the network having knowledge of or be required to handle security services for the packet. Tunnel mode may also be used by a security gateway that

provides security services for a Virtual Private Network. In this arrangement, the cryptographic endpoint is listed in the outer IP header (the peer that will provide the security services for the hidden network), while the communications endpoint is identified in the inner header, and is the one sitting behind the gateway.

Transport mode is used when the cryptographic and communications endpoints are the same.

Figure 5 shows an ESP packet in Tunnel mode and in Transport Mode [1], [6].

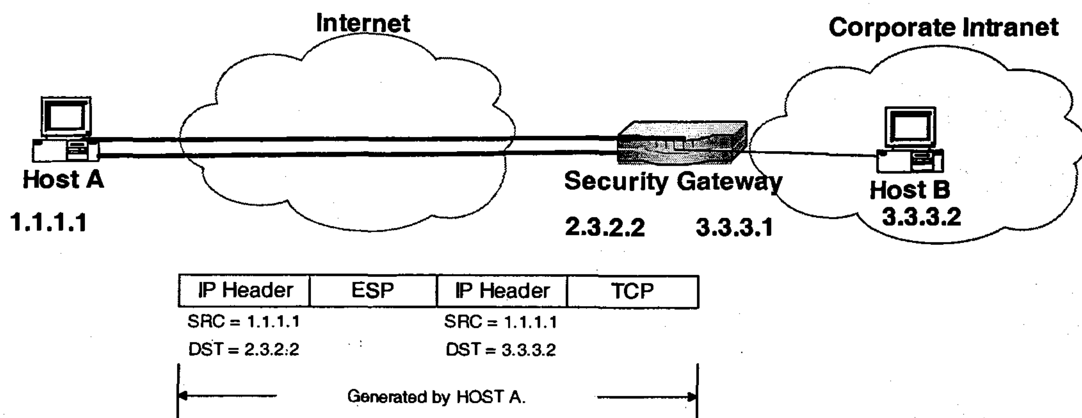
Figure 5 ESP Packet in Tunnel and Transport Mode



As an example of a tunnel mode arrangement, consider Figure 6. In this case, Host A wishes to communicate securely with Host B that is inside a corporate Intranet. Therefore, it must establish a secure connection with the Intranet's security gateway. As Figure 6 shows, Host A generates a packet with an IP header indicating the destination address of the host within the corporate network. This IP packet is then encapsulated

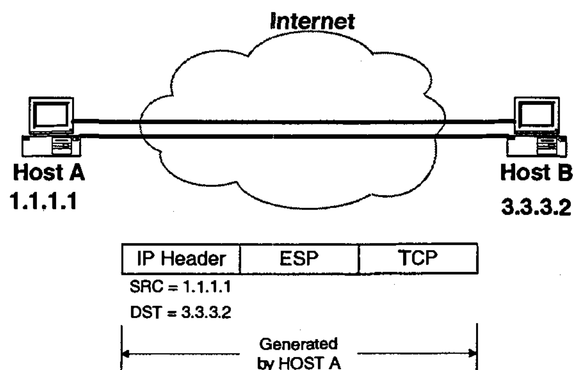
with ESP in tunnel mode. The outer IP header is used to route the packet through the Internet to the security gateway. Once the security gateway receives the packet, it realizes that it is the destination for the ESP packet and performs inbound IPSec processing on it before forwarding it within the corporate network. This example is typical of the remote user-to-corporate VPN connection.

Figure 6 Tunnel Mode Example



A transport mode example is shown in Figure 7. This example illustrates a situation where the communications endpoint is also the cryptographic endpoint.

Figure 7 Transport Mode Example



3.2 IPSec Security Associations and Policy

The IPSec protocols together indicate what packets to protect, how to protect them, and with whom the protection is shared. This information is maintained on a peer to peer basis by way of a Security Association (SA) which is stored in the SA Database. An SA is a unidirectional element that maintains the state of the secure link. Each peer must maintain two SAs for every end-point to which secure communications are desired. Among other things, the SA indicates the keys to be used with the encryption and authentication algorithms, the lifetimes of the keys (all keys must expire at some point otherwise security is undermined), the sequence number (for replay protection) as well as other context information [1], [6], [7].

As noted previously, the SPI contained in the ESP and AH packets along with the source and destination addresses, and IPSec protocol are used as indexes into the SADB. Another database, the Security Policy Database (SPD) is used to indicate what processing should take place with a given packet, including whether or not security services need to be applied, what security protocol (ESP, AH) to use, and in what mode, and what encryption/authentication algorithms to use (DES, AES, HMAC-MD5, etc.).

If policy indicates that security services need to be applied, but no SA exists, the Internet Key Exchange (IKE) is used to establish the SAs which must be in place to allow traffic to flow. As part of this process, the keys used by the encryption algorithms such as AES and 3-DES are established [1], [6].

3.3 IPSec Processing

The following sections describe the basic steps that are followed in the inbound and outbound direction for ESP packets.

3.3.1 Inbound Processing

Upon receipt of an IP packet, the receiver performs the following [1]:

1. Determines whether an SA exists for the packet. If none exists, the packet is dropped.

2. Assuming an SA exists, the sequence number is then processed to ensure that it is valid and not a potential replay packet.

3. The packet is then authenticated using the specified authentication algorithm and key. The generated authentication result is then compared with the authentication data in the header. If equal, processing proceeds.

4. The packet is then decrypted using the specified decryption algorithm and key. The decrypted result is checked for accuracy (usually using the pad for verification purposes).

5. The mode of the packet is then validated against what is expected (tunnel and/or transport) in the SA and policy. If not correct, the packet is dropped.

6. The IP packet is then re-built, with the ESP header extracted. The port and protocol of the packet is then validated against policy.

7. Finally, assuming all checks have passed, the IP packet is forwarded to the IP processing engine which determines the next steps for the packet (whether this is the destination, or whether it needs to be forwarded to the next hop).

3.3.2 Outbound Processing

Before a packet can be transmitted, the following outbound processing is performed [1]:

1. An ESP header is inserted in the proper location for tunnel or transport mode.
2. The appropriate packet fields are encrypted.
3. The appropriate packet fields are authenticated, and the authentication result is placed in the authentication data field of the ESP trailer.
4. The IP header checksum is re-computed (if necessary).

CHAPTER 4 ENCRYPTION ALGORITHMS

There are two general classes of encryption algorithms [5]:

- Symmetric key
- Public-key

A symmetric key encryption algorithm is one in which both ends of an encrypted conversation use the same key, for both encrypting and decrypting the data. In other words, both parties in the conversation must know the key. However, this raises the important issue of key distribution. If one party wants to use a specific key, how do they let the other party know the key to use? They could not simply transmit the key to the other party, as this allows any person with access to the transmission to receive all subsequent data transmitted using that key, thus defeating the purpose of encrypting data in the first place. Nor could the key be mailed, telephoned, or faxed to the far end as all these methods are both insecure, and non-scalable.

The solution to this problem is to use public-key cryptography. In public-key cryptography, two keys are used [5]:

- Public key: can only be used to encrypt data
- Private key: can only be used to decrypt data

Typically, a users public-key is stored in a public database such as a Certificate Authority. When user A needs to send a message to user B, user A retrieves B's public key from the database, and encrypts the message using the public key. User B can then decrypt the message using his private key (which only he has access to).

Public-key encryption algorithms are typically used as part of the key distribution process for the symmetric key algorithm. The public key algorithm is used to encrypt the key for the symmetric key algorithm prior to transmission to the far end peer. Upon receiving the message, the far end peer decrypts the symmetric key using his private key.

One may wonder why it is necessary to use two different encryption algorithms, when public-key cryptography can be used to encrypt data, and elegantly solves the key distribution problem. The reason is that public key encryption algorithms rely heavily on modular exponentiation using large integers [6], which is very computationally intensive and slow. In fact, public key encryption algorithms can be three orders of magnitude or more slower than a symmetric key algorithm. For this reason, symmetric key encryption algorithms are used to protect data, while public key encryption algorithms protect the key to be used by the symmetric key algorithm [5].

A further classification of symmetric key algorithms is whether the encryption algorithm (cipher) operates on a fixed sized block of data at a time (block cipher), or on a single bit at a time (stream cipher) [5]. The Data Encryption Standard (DES) and its replacement, the Advanced Encryption Standard (AES), are examples of block ciphers, and the focus of this Thesis.

4.1 Modes of Operation

All block based symmetric key encryption algorithms can be operated in one of two principal modes [5]:

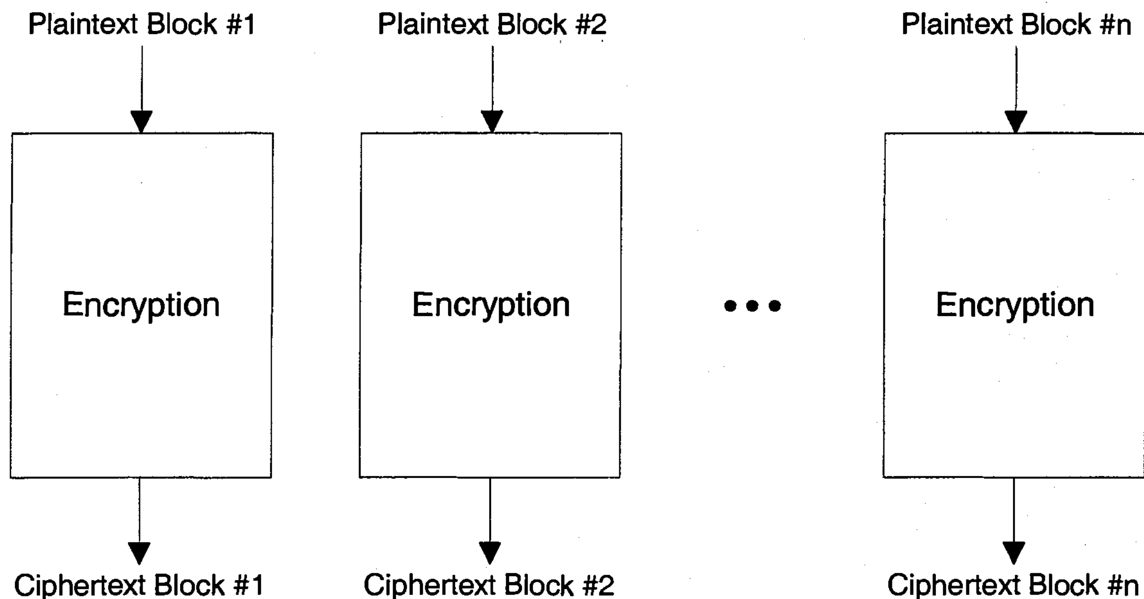
1. Electronic Code Book (ECB)
2. Cipher Block Chaining (CBC)

Other modes, such as Cipher-Feedback (CFB) and Counter (CTR) are possible, though they are not as commonly implemented, and therefore are outside the scope of this Thesis.

4.1.1 Electronic Code Book (ECB) Mode

Electronic Code Book mode is the simplest way to operate a block cipher. Blocks of plaintext are simply run through the cipher without any additional feedback from previous encryption rounds. In ECB mode, a block of plaintext always encrypts to the same block of ciphertext (assuming the key is the same) [5]. Figure 8 shows 'n' blocks of plaintext encrypting to 'n' blocks of ciphertext. Note that for decryption, a similar drawing can be made, with ciphertext block #1 decrypting to plaintext block #1 and so on.

Figure 8 Electronic Code Book mode



Unfortunately, due to its simplicity, ECB mode is susceptible to attack. Messages transmitted on the Internet tend to follow a defined format due to the need to abide by

various networking protocols, such as IP or TCP. For instance, the messages will all likely have an IP header, which has a predefined format that includes certain fields that either don't change or don't change very often for a particular user, such as an IP source address. If an attacker is able to gain access to IP packets transmitted using ECB mode, they will quickly be able to determine what key was used during transmission. The attacker will attempt decryption of the packet using all of the possible different keys, but only some of these attempts will yield reasonable, usable results. All others will be discarded. For instance, a decryption that yields an IP source address field of 7k*.p@n.uYS.98# will be discarded, while one that yields 233.140.70.4 will be accepted. In the first case, that value cannot possibly form an IP Source Address, so that key attempt is obviously incorrect. While the second value could be an IP Source Address, which means that the key attempted may in fact be the actual key used to transmit the data. Once a key is known, the attacker could theoretically do anything he wished to the communication, from simply snooping, to injecting false packets.

Another weakness of ECB mode is that it is susceptible to an attack known as block replay [5]. A block replay attack uses the fact that a block of plaintext always encrypts to the same block of ciphertext. Using this knowledge, an attacker simply replays certain blocks of the message multiple times.

One advantage of ECB mode over the other modes is that since no feedback is involved, the encryption and decryption process can be easily parallelized and pipelined.

4.1.2 Cipher Block Chaining Mode

CBC mode avoids the security holes found in ECB mode by applying feedback to the encryption and decryption process. The same block of plaintext will no longer encrypt

to the same block of ciphertext. With CBC, the encryption of plaintext block 'n' depends on the encryption of plaintext blocks 1 through n-1.

Figure 9 depicts the process. Plaintext block #1 is XOR'ed with an Initialisation Vector (IV) before being encrypted. An Initialisation Vector is some random value that is used to kick-start the encryption (decryption) process for the block. All subsequent blocks of the same message are XOR'ed with the ciphertext result of the previous block. For instance, plaintext block #2 is XOR'ed with ciphertext block #1, and so on until the end of the message is reached [5]. Note that the encryption of plaintext block #2 can not commence until the encryption of plaintext block #1 completes.

Figure 9 Cipher Block Chaining mode, Encryption

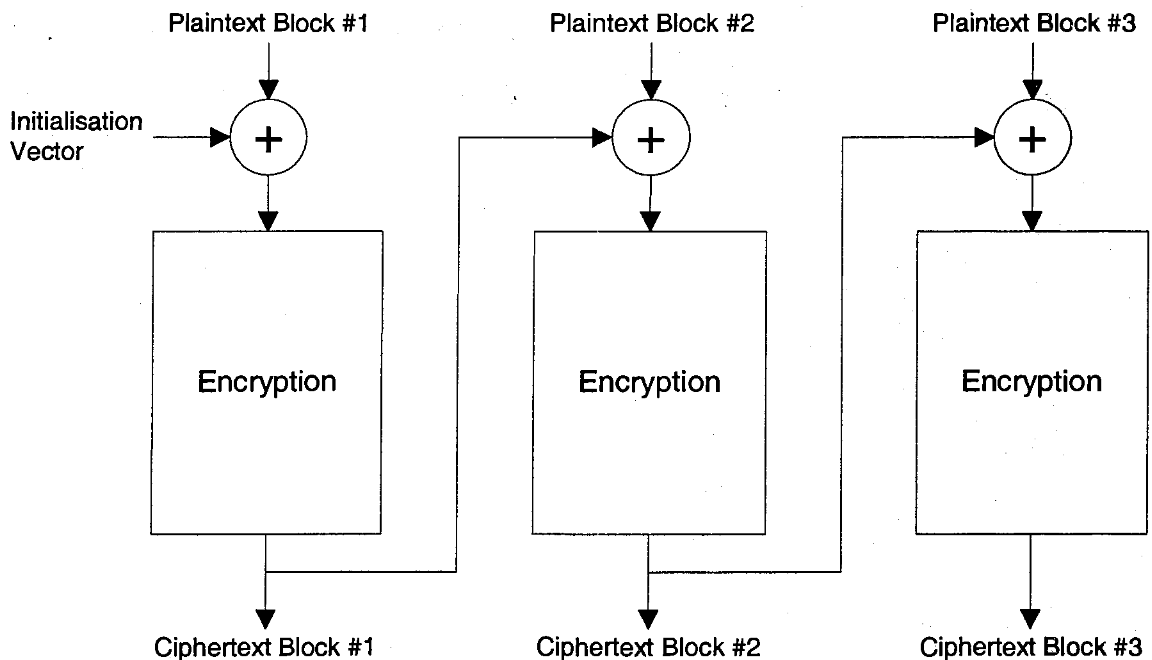
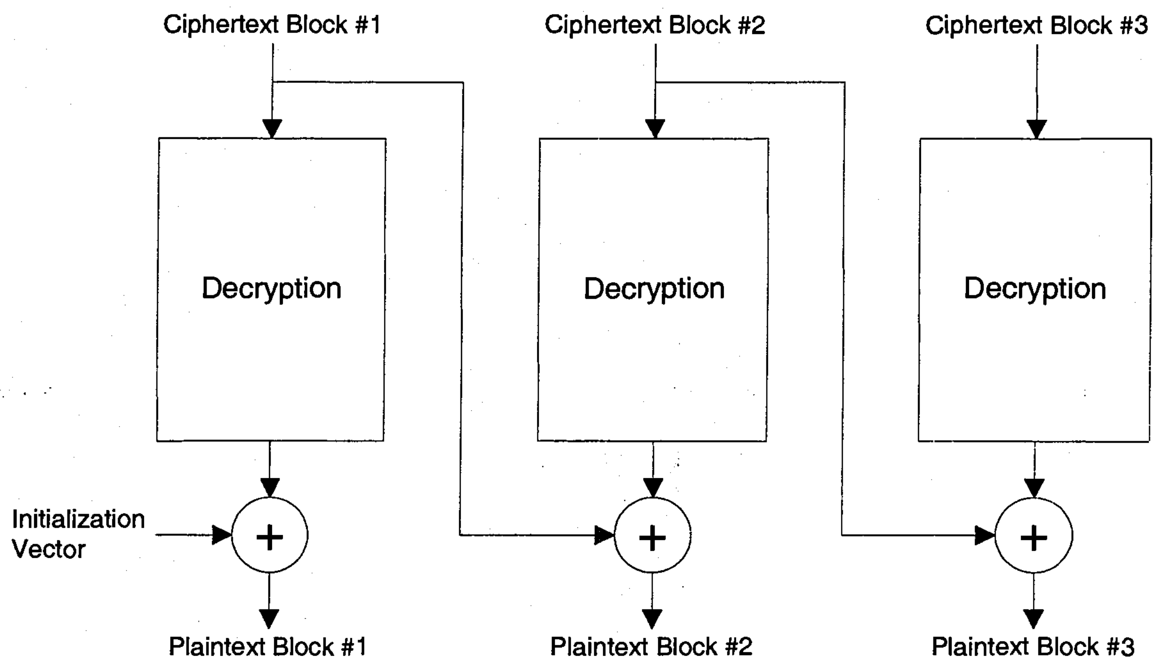


Figure 10 depicts how CBC mode works when decrypting data. Here, after the first block of data is processed by the symmetric key algorithm, the result is XOR'ed with the IV, creating plaintext block #1. For all subsequent blocks of the message, plaintext block

#n is found by processing ciphertext block #n with the symmetric key algorithm, and XOR'ing the result with plaintext block #n-1. Unlike encryption using CBC mode, decryption can be easily parallelized, allowing decryption of blocks 2 onwards to begin before block 1 completes.

Figure 10 Cipher Block Chaining mode, Decryption



The following two sections describe the design of the two most common symmetric key algorithms, DES and AES.

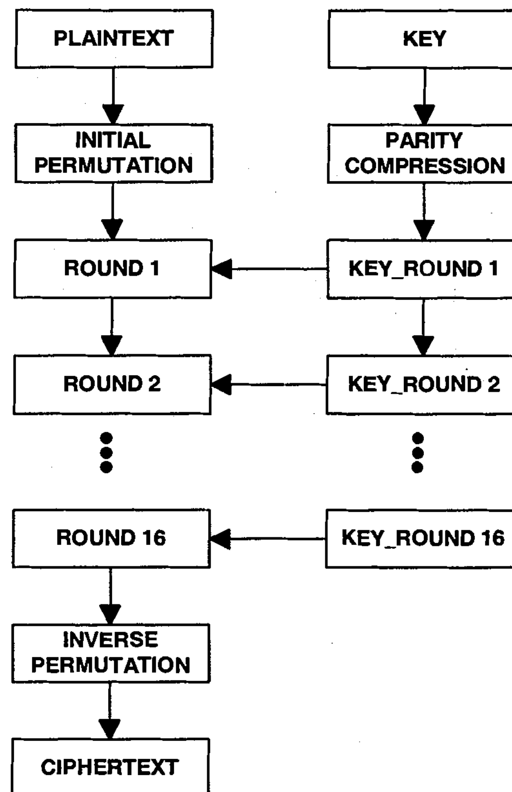
4.2 The Data Encryption Standard (DES)

DES [10] was adopted as a U.S. Federal Government standard for encryption in 1976, and by ANSI for use in the private sector in 1981 [5]. DES is an iterative block cipher, that uses a block size of 64 bits, and a key size of 64 bits (although every 8th bit of the key is a parity bit).

Figure 11 shows the high level structure of the DES algorithm, along with its key expansion process [5].

As can be seen, a block of plaintext first goes though an initial permutation block. The data is then cycled through the same round function 16 times (each time using a new key from the key expansion process) before going through the inverse permutation block.

Figure 11 DES Structure



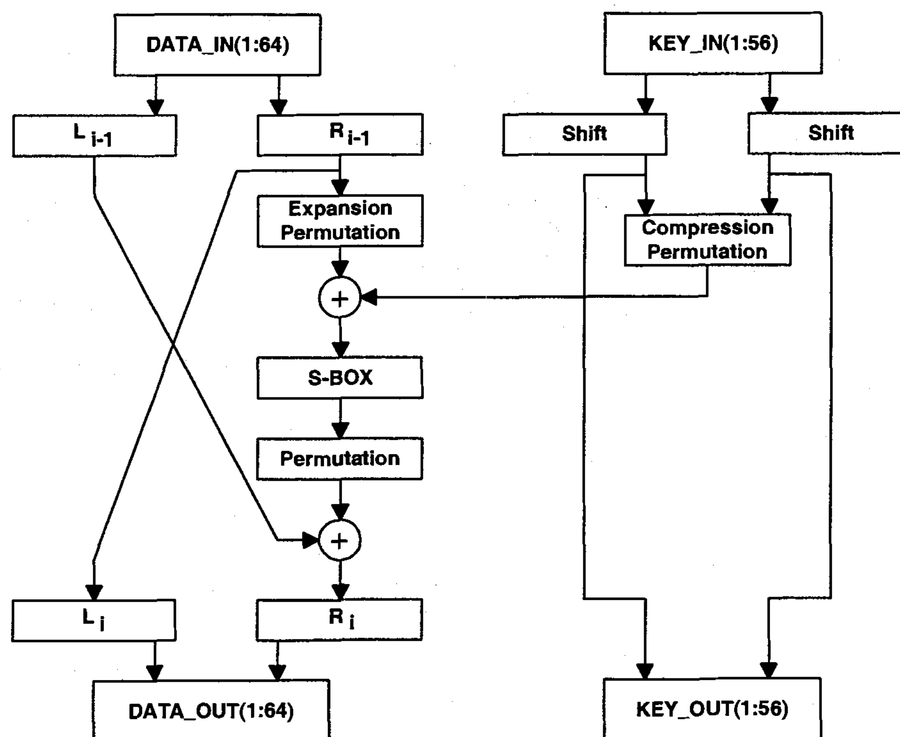
All 16 rounds of DES have the structure shown in Figure 12 [5], [10]. The input data is split into two halves, a left half and a right half. The right half of the data goes through an expansion permutation that expands the data from 32 bits to 48 bits. The data is then XOR'ed with the specific key for this particular round before being passed to the input of

the S-BOXs. An S-BOX is a non-linear replacement of one value with another. DES uses 8 S-BOXs that each take 6 bits as input, and produce a 4 bit output. Therefore, after the S-BOX function is performed, the data is again 32 bits wide.

Following the S-BOX replacement, the 32 bit data is once again permuted, and then XOR'ed with the left half of the initial input data. The end result is a new 32 bit string for the right half of the data.

The left half output is simply equal to the right half input data.

Figure 12 DES Round Function



As noted previously, the DES algorithm uses an initial key size of 64 bits. Each round of the algorithm uses a different 48 bit round key that is based on the initial input key. In other words, the initial input key is used to create 15 additional keys to be used for

rounds 2 through 16. The initial input key and the 15 additional round keys are collectively known as the key schedule, and are created through a key expansion process.

As an initial step of the key expansion process, the 64 bit input key is reduced to 56 bits by removing (and checking) the parity bits. The 56 bit key is then divided into left and right 28 bit halves, and circularly left shifted by one or two bits. The round number is used to determine how many bits (1 or 2) to shift by. Following the shift, the key is compressed and permutated to 48 bits by the compression permutation. The output of the compression permutation serves as the key to be used for this round, while the output of the circular shift serves as the input to the next round's key expansion process.

Note that the above discussion was focussed on the encryption case. For decryption, the exact same high level and round structures can be used. The only differences are that keys are used in reverse, the keys are expanded using right shifts, and the keys are shifted a different number of times than in the encryption case.

Specific details of the permutations and the contents of the S-boxes are given [10].

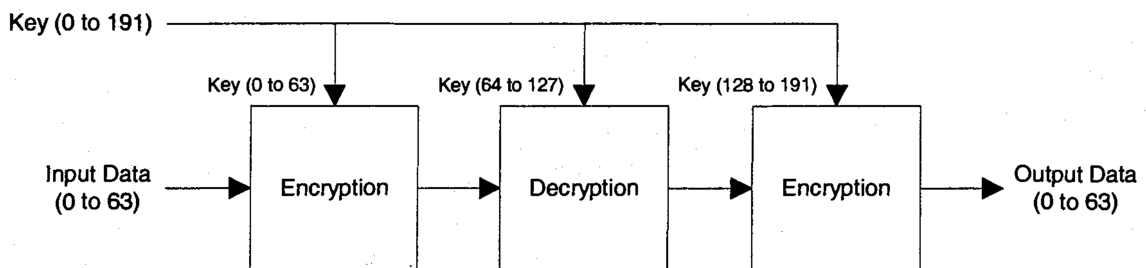
4.2.1 Triple-DES (3-DES)

As computational power has increased over the years, so to has the ability of hackers to break DES. Due to its short, 56 bit key space, DES can be cracked. Triple-DES was introduced to address this problem.

Triple encryption is a general technique that can be applied to any symmetric key algorithm [5]. The end result is increased security via a larger key (Triple-DES uses a 192 bit key). The basic idea is illustrated in Figure 13.

Here, an encryption operation is first applied to the data using bits 0 to 63 of the key. The ciphertext output of the first encryption operation is then fed into a decryption process that utilizes bits 64 to 127 of the key. Finally, the output of the decryption block is fed into final encryption block that uses bits 128 to 191 of the key. For 3-DES, the result is a 48 round process.

Figure 13 Triple-DES



4.3 The Advanced Encryption Standard

As a result of the dual needs for increased security, and for an algorithm that can be implemented efficiently with high throughput in hardware or software, the U.S. National Institutes of Standards and Technology (NIST) launched a formal competition to define a replacement algorithm for DES. After a lengthy evaluation process, the RIJNDAEL algorithm was standardized as the new AES in 2001 [11].

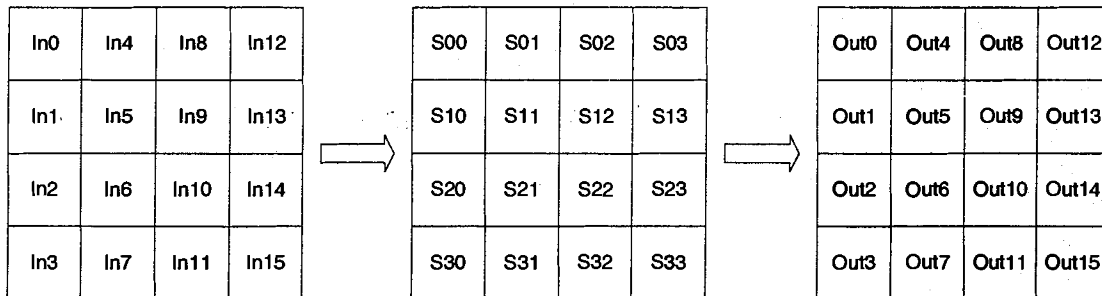
AES is a symmetric key block cipher that uses a block size of 128 bits, and a key size of 128, 192, or 256 bits [11]. The algorithm is iterative, requiring 11, 13, or 15 rounds (depending on the key size) to produce an output. Unlike DES, the same exact operations cannot be performed for both encryption and decryption.

The following sections describe the AES cipher, inverse cipher, and key expansion in further detail [11].

4.3.1 AES Cipher

Initially (and for all rounds that follow), the data to be processed by the cipher is organized into a 4 x 4 matrix called the State. Each element of the State corresponds to one of the bytes in the input data block. As shown in Figure 14, input byte 0 (bits 0 to 7 of the input data block), corresponds to the element at row 0, column 0. The cipher processes the bytes and columns of the state to produce the output state.

Figure 14 The AES State



A high level view of the AES Cipher is shown in Figure 15. In the initial round (round 0), the plaintext is simply XOR'ed with the input key. The result is then iteratively processed by the round function for rounds 1 through N. In round N+1, a modified round function (minus the MixColumns operation) is applied to the output of round N to create the ciphertext output.

As shown in Figure 16, each of rounds 1 through N in Figure 15 contain the following four operations:

1. SubBytes

2. ShiftRows
3. MixColumns
4. AddRoundKey

The following sections describe each of these operations in further detail.

Figure 15 AES Cipher

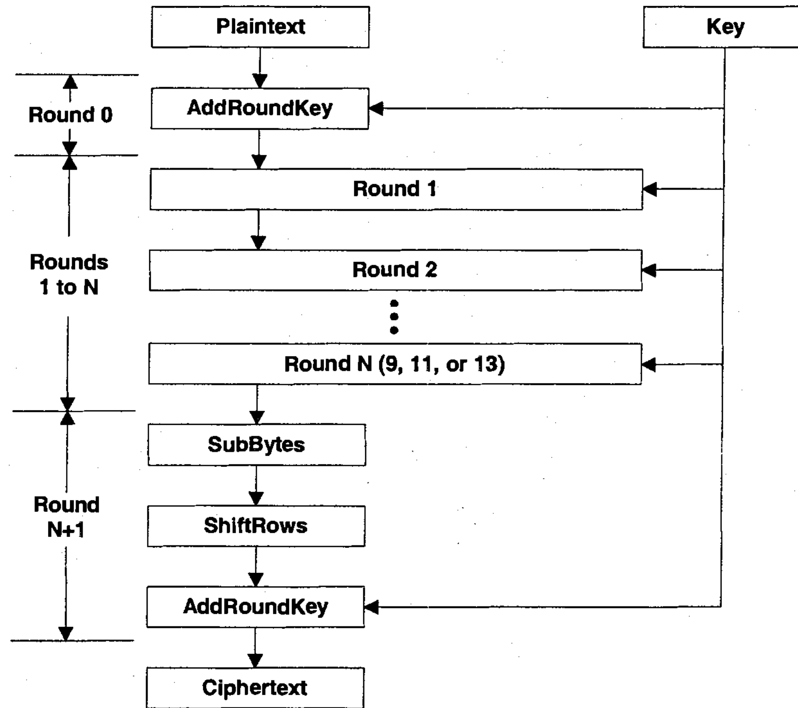
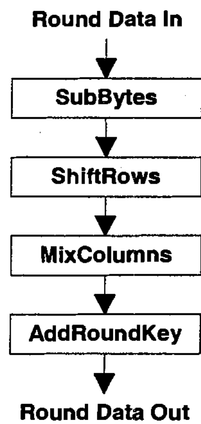


Figure 16 AES Round Function



4.3.1.1 SubBytes

SubBytes is a non-linear byte substitution of each individual byte of the State. There are essentially two approaches for implementing the SubBytes process:

1. Use a look-up table
2. Perform the following calculation, where b_n' is the result of transforming b_n.

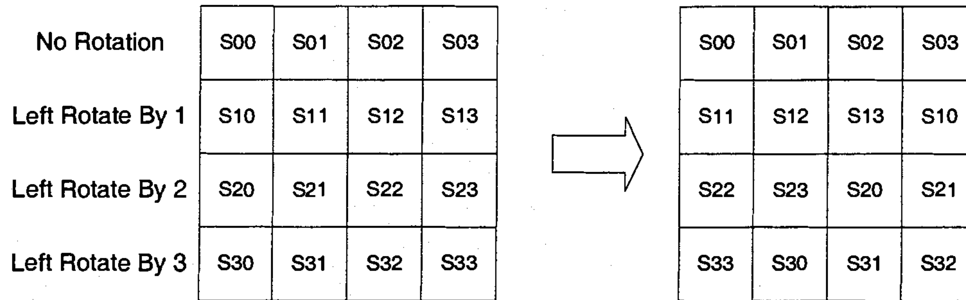
$$\begin{bmatrix} b_0' \\ b_1' \\ b_2' \\ b_3' \\ b_4' \\ b_5' \\ b_6' \\ b_7' \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \\ b_4 \\ b_5 \\ b_6 \\ b_7 \end{bmatrix} + \begin{bmatrix} 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \\ 0 \end{bmatrix}$$

As noted above, a look-up table that implements the SubBytes transformation can easily be found by simply plugging in all 256 possible bytes into the above matrix.

4.3.1.2 ShiftRows

ShiftRows applies a variable number of circular left shifts over each row of the state. Each row is shifted an amount given by its row number. For instance, row 0 is not shifted, row 1 is shifted 1 position, and so forth. Figure 17 graphically depicts the ShiftRows process.

Figure 17 ShiftRows



4.3.1.3 MixColumns

The MixColumns transform multiplies each column of the State by a fixed matrix to produce a new column. The following equation describes the multiplication:

$$\begin{bmatrix} S'_{0,C} \\ S'_{1,C} \\ S'_{2,C} \\ S'_{3,C} \end{bmatrix} = \begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix} \begin{bmatrix} S_{0,C} \\ S_{1,C} \\ S_{2,C} \\ S_{3,C} \end{bmatrix}$$

yielding the following set of equations, where C indicates the column number:

$$S'_{0,C} = 2 * S_{0,C} + 3 * S_{1,C} + 1 * S_{2,C} + 1 * S_{3,C}$$
$$S'_{1,C} = 1 * S_{0,C} + 2 * S_{1,C} + 3 * S_{2,C} + 1 * S_{3,C}$$
$$S'_{2,C} = 1 * S_{0,C} + 1 * S_{1,C} + 2 * S_{2,C} + 3 * S_{3,C}$$
$$S_{3,C} = 3 * S_{0,C} + 1 * S_{1,C} + 1 * S_{2,C} + 2 * S_{3,C}$$

The 1x multiplication is trivial, as the result is simply the input byte. The 2x multiplication can be realized by multiplying the value (for example $S_{0,C}$) by 2, and checking whether the initial value ($S_{0,C}$) is > 127. If so, subtract (using bitwise XOR) 0x1B. If not, the result is already in final form. The 3x multiplication is also trivial, as it simply is the addition (using bitwise XOR) of the 1x and 2x values.

4.3.1.4 AddRoundKey

AddRoundKey simply XORs the State with the particular key for the round.

4.3.2 AES Inverse Cipher

The AES Inverse Cipher has a similar overall structure to the AES Cipher. The primary differences are that the transforms are the inverse of those used in the AES Cipher, the keys are used in reverse order (thus the round ordering is reversed), and the specific order of operation of the individual transforms is altered slightly, as shown in Figure 19.

Figure 18 AES Inverse Cipher

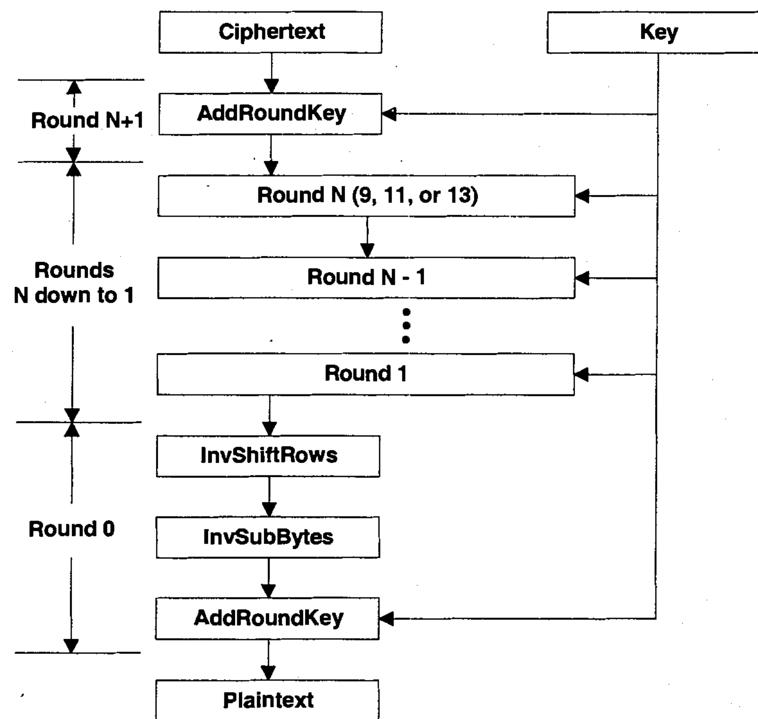
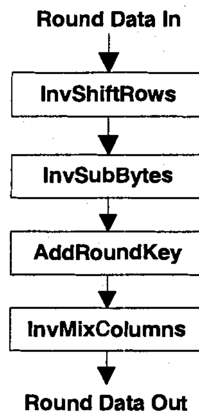


Figure 19 AES Inverse Cipher Round Function

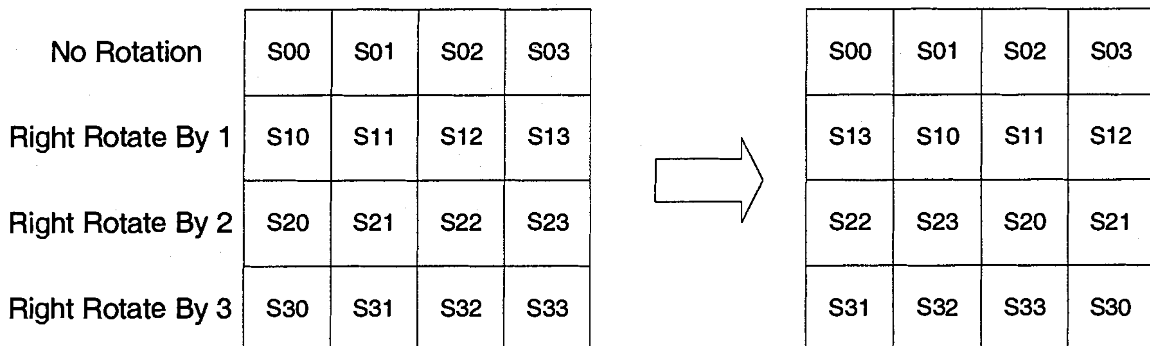


The following sections describe the functionality of the InvShiftRows, InvSubBytes, and InvMixColumns transforms. The AddRoundKey operation is identical to that described in section 4.3.1.4.

4.3.2.1 InvShiftRows

InvShiftRows applies a circular right shift to each row of the state. The number of positions each row is shifted depends on the row number, as illustrated in Figure 20.

Figure 20 InvShiftRows



4.3.2.2 InvSubBytes

The InvSubBytes operation is the inverse of the SubBytes procedure. Therefore, an inverse look-up table can be created to perform this procedure. As an example, an input byte of 0x00 to the SubBytes procedure will yield an output value of 0x63. Therefore, for the InvSubBytes look-up table, the value obtained with an input of 0x63 should be 0x00.

4.3.2.3 InvMixColumns

Like the MixColumns operation, the InvMixColumns transform multiplies each column of the State by a fixed matrix to produce a new column. The following equation describes the multiplication:

$$\begin{bmatrix} S'_{0,C} \\ S'_{1,C} \\ S'_{2,C} \\ S'_{3,C} \end{bmatrix} = \begin{bmatrix} 0E & 0B & 0D & 09 \\ 09 & 0E & 0B & 0D \\ 0D & 09 & 0E & 0B \\ 0B & 0D & 09 & 0E \end{bmatrix} \begin{bmatrix} S_{0,C} \\ S_{1,C} \\ S_{2,C} \\ S_{3,C} \end{bmatrix}$$

yielding the following set of equations, where C indicates the column number:

$$S'_{0,C} = E * S_{0,C} + B * S_{1,C} + D * S_{2,C} + 9 * S_{3,C}$$
$$S'_{1,C} = 9 * S_{0,C} + E * S_{1,C} + B * S_{2,C} + D * S_{3,C}$$
$$S'_{2,C} = D * S_{0,C} + 9 * S_{1,C} + E * S_{2,C} + B * S_{3,C}$$
$$S'_{3,C} = B * S_{0,C} + D * S_{1,C} + 9 * S_{2,C} + E * S_{3,C}$$

The values 09x, 0Bx, 0Dx and 0Ex are obtained through successively applying the multiplication approach described in section 4.3.1.3. For example, the 9x multiplication is obtained by multiplying by 2x three times, and adding the 1x value.

4.3.3 AES Key Expansion

As noted previously, the AES Cipher and Inverse Cipher require a new key value to be used for each round. Figure 21 depicts the key expansion operation for a key size of 128

bits. The input key is split into 4 32 bit words (words 0 through 3). The RotWord process takes a 4 byte word (b0, b1, b2, b3) and performs a byte permutation to yield (b1, b2, b3, b0). Each byte of the word is then replaced using the same S-BOX as described in the SubBytes transform. The word is then XOR'ed with a constant value that is based on the round number, as shown in the following table:

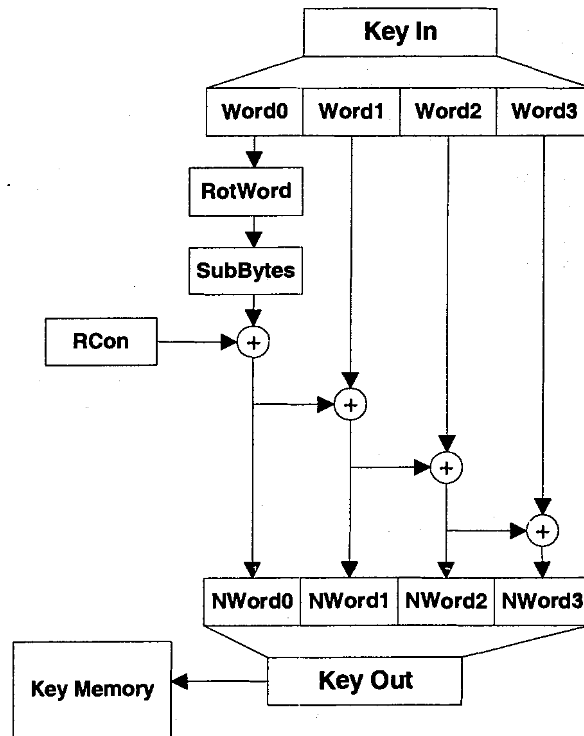
Table 1 Round Constant (RCON) Values for Key Expansion

Round	RCON Value
1	0x01000000
2	0x02000000
3	0x04000000
4	0x08000000
5	0x10000000
6	0x20000000
7	0x40000000
8	0x80000000
9	0x1B000000
10	0x36000000

The result of the RCON operation forms word0 of the next key (NWord0). This particular value is also XOR'ed with word1 of the input key to create NWord1. Word2 is XOR'ed with NWord1 to create NWord2. Word3 is XOR'ed with NWord2 to create NWord3.

This process creates 10 new key values from the initial 128 bit input key, for a total of 11 round keys. These round keys can be stored in memory to be used as necessary.

Figure 21 AES Key Expansion



CHAPTER 5 DESIGN AND IMPLEMENTATION OF AES

The design and implementation of AES typically involves making tradeoffs of processing speed vs. area/power. While some applications such as an Internet core router would require the fastest possible implementation, other applications such as wireless PDAs would be more concerned about minimizing area and power consumption.

Design architecture, algorithm implementation, and implementation form factor (FPGA, ASIC etc.) are three situations where one must be cognizant of the end goal (e.g. highest possible throughput, lowest possible power or some optimisation in between).

The ways in which the design architecture, algorithmic implementation, and form factor affect speed and area/power are discussed generally before describing the actual VHDL implementations chosen for this Thesis.

5.1 Architectural Options

The three most common architectures typically employed when implementing a block cipher, such as AES, in hardware [12] are:

- Pipelining
- Sub-pipelining
- Loop Unrolling

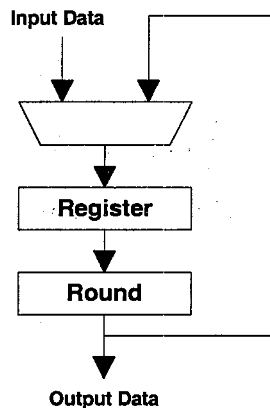
This Thesis proposes a fourth approach, termed Multi-Session Pipelining, which seeks to apply the benefits of pipelining to CBC mode in a novel way.

5.1.1 Pipelining

In pipelining, registers are inserted between each round that forms the pipeline. The depth of the pipeline, K, determines how many data blocks can be processed simultaneously. The architecture is fully pipelined when K equals the number of rounds, N [12].

Note that with K=1, the architecture becomes that shown in Figure 22. This is the smallest possible implementation of an N-round algorithm [12].

Figure 22 Pipeline architecture with K = 1



Pipelined architectures are suitable and offer the highest performance for ciphers operating in non-feedback modes such as ECB, where each block of data is encrypted (or decrypted) independently of one another. Zhang [12] shows that for non-feedback modes, both speed and area increase by a factor of K for pipelined architectures over the basic architecture shown above. Much has been written about extremely fast pipelined implementations of AES [20] – [24].

However, pipelined architectures are not quite so suitable for ciphers operating in feedback mode (such as CBC). This is because all rounds of the algorithm must be

performed on data block 'N' before data block 'N+1' from the same packet (and using the same key) can be processed (due to the block chaining that is in effect). Section 5.1.4 of this Thesis discusses how the pipeline and the external data source can be modified to enable pipelined architectures to improve the aggregate throughput of the Cipher when operating in CBC mode. Section 6.2 of this Thesis discusses an AES Cipher design based on this Multi-Session Pipelined approach.

5.1.2 Sub-Pipelining

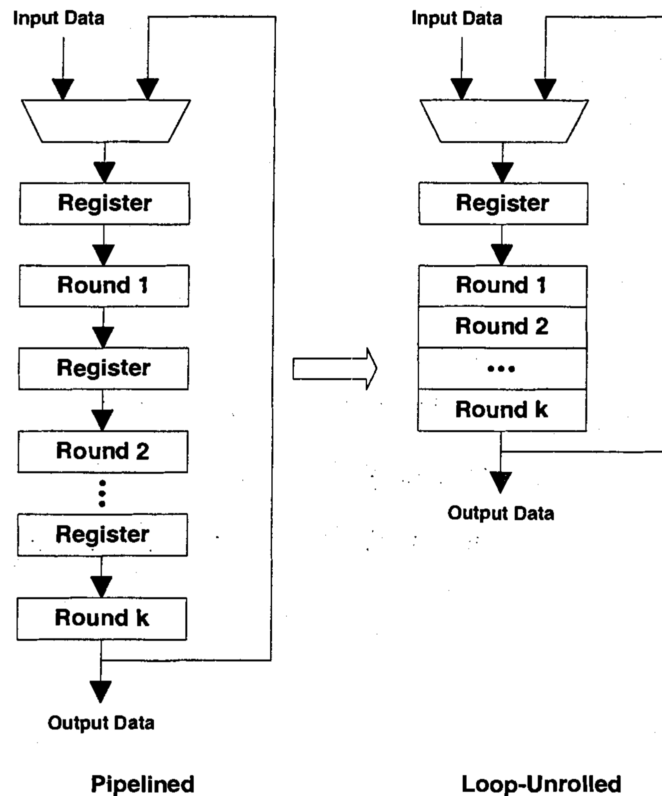
In sub-pipelining, registers are actually inserted inside the round function itself, thereby essentially splitting the round function into two sections. This is essentially the same concept as pipelining. For non-feedback modes of operation, Zhang et al. [12] shows that a sub-pipelined architecture with each round divided into r=2 sections offers a 2*K improvement in throughput (with K equal to the depth of the main pipeline). The additional throughput comes at a cost of k*(r-1) additional registers for the sub-pipelining functionality. Note that as with standard pipelined architectures, sub-pipelining is generally not suitable for ciphers operating in a feedback mode. In fact, sub-pipelining may degrade performance when used with CBC mode.

5.1.3 Loop Unrolling

In loop unrolling, the basic architecture of Figure 22 is modified by inserting additional rounds of combinatorial logic inside the loop, but without the additional expense of registers. In this architecture, multiple rounds of the algorithm are processed in the same clock cycle. Since the delay of each round (assumed to be due to combinatorial logic) is fixed, the clock period must increase to ensure the data is processed by each round in the same clock cycle. Unlike the pipelined architecture where registers are

inserted between each round, these inter-round registers are not present in the loop-unrolled architecture. Figure 23 shows the difference between a pipelined and loop-unrolled architecture.

Figure 23 Pipelined vs. Loop Unrolled Architectures



In a loop-unrolled architecture, throughput is increased by eliminating the delay associated with the pipeline register(s) [12]. If one assumes the minimum clock period for the basic architecture (pipeline with K=1) is as follows:

$$T_{ARCH} = T_{ROUND} + T_{OH} ,$$

where T_{ROUND} is the delay associated with the actual round function processing, and T_{OH} is the overhead delay (setup and propagation) associated with the register(s) and multiplexers of the chosen architecture. Throughput [12] is then:

$$Throughput_{BASIC(K=1)} = \frac{128}{NR * T_{ARCH}} = \frac{128}{NR * T_{ROUND} + NR * T_{OH}},$$

where NR is the number of rounds to be processed, and 128 is the number of output bits produced. For AES operating with 128 bit keys, NR = 10.

Note that the throughput for a fully pipelined (K=11) design operating in ECB mode is:

$$Throughput_{PIPE(K=11)} = \frac{128}{T_{ARCH}} = \frac{128}{T_{ROUND} + T_{OH}}.$$

To calculate the throughput improvement achievable using loop-unrolling, the delay must first be calculated. This is derived from the following, where K indicates the number of rounds processed in the same clock cycle:

$$T_{ARCH} = K * T_{ROUND} + T_{OH}.$$

Throughput can now be expressed as:

$$Throughput_{LU} = \frac{128}{\frac{NR}{K} * T_{ARCH}} = \frac{128}{NR * T_{ROUND} + \frac{NR}{K} * T_{OH}}.$$

The speedup achieved by using loop-unrolling can be determined by solving the following equation:

$$SPEEDUP = \frac{Throughput_{LU}}{Throughput_{BASIC}},$$

to yield:

$$SPEEDUP = \frac{1+\tau}{1+\dfrac{\tau}{K}}$$

where $\tau = T_{OH}/T_{ROUND}$. The following table shows the magnitude of the speedup that can be achieved by using the loop-unrolling method. If one assumes the overhead processing delay is 40% of the round processing delay (for a total delay of 14 units), a fully loop-unrolled architecture where K=10 will only experience a 35% speedup over the basic architecture. Though significant, this throughput increase will come at the cost of increased area, on the order of K times that of the basic architecture.

Only K values of 1, 2, 5, and 10 are suitable for the AES algorithm when using 128 bit keys. Table 2 also shows that for constant τ and as K increases, the rate of throughput increase diminishes. In this example, K=2 appears to present the greatest throughput increase versus area trade off. Recall that the throughput increase of the pipelined architecture was nearly K times that of the basic architecture.

Table 2 Speedup achieved by using loop-unrolling

Toh	1	4	9	4	4	4	4
Tround	10	10	10	10	10	10	10
Tau	0.1	0.4	0.9	0.4	0.4	0.4	0.4
k	10	10	10	1	2	5	10
SPEEDUP	**1.09**	**1.35**	**1.74**	**1.00**	**1.17**	**1.30**	**1.35**

The advantage of loop-unrolling is that it is applicable to CBC and other feedback modes of operation.

5.1.4 Multi-Session Pipelining

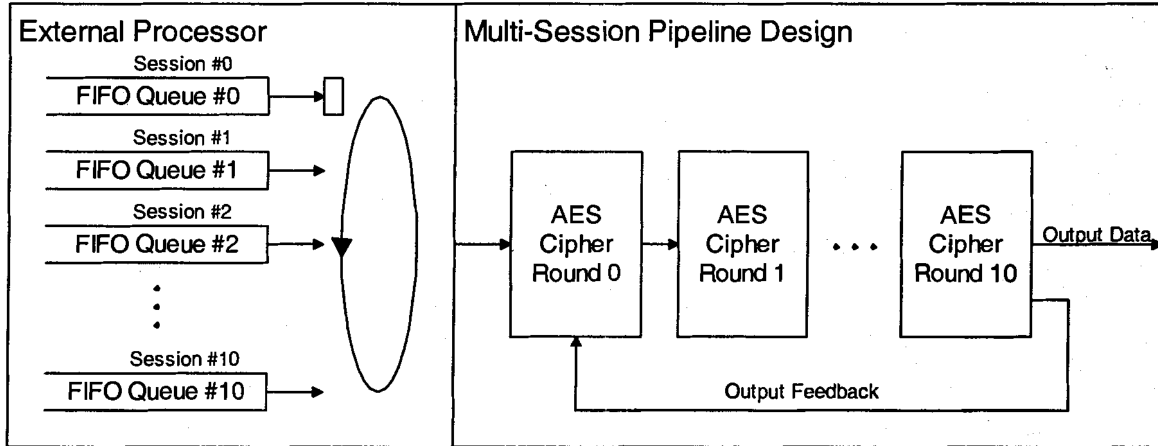
As mentioned previously, the primary drawback of the standard pipeline approach is that it is not well suited to feedback-based modes of operation (such as CBC) due to the need to complete the encryption of block N before block N+1 from the same packet can be encrypted. [20] describes a method of processing four concurrent 32 bit threads at a time in order to increase throughput, however, it appears that this approach does not support CBC mode. Multi-Session Pipelining is a novel method proposed in this Thesis for extending the benefits of pipelining to CBC mode.

An important observation is that blocks of data from other packets (using other keys) could be used to fill the pipeline. Each of the distinct user and key combinations to which encryption services are being applied are referred to as a session.

In order to allow this Multi-Session Pipeline approach to work for CBC mode, the scheduling of block data into the pipeline must be modified to ensure that blocks of data from the same session are always input to the Cipher NR rounds apart, where NR is 11 (owing to the depth of the pipeline, and the number of rounds in the AES Cipher). This is accomplished by maintaining NR distinct queues and servicing the queues in a round-robin fashion.

In addition, a feedback path must be created from the output of the last round to the input of the first round of the algorithm. Figure 24 on the next page shows the proposed architecture.

Figure 24 Multi-Session Pipeline System Diagram



The External Processor depicted in Figure 24 is designed to handle up to 11 FIFO queues, which are serviced in a strict round-robin fashion. Data from the same session is always placed into the same queue. More than 11 sessions may be supported by populating the queues with multiple sessions, so long as all blocks of data corresponding to a particular packet/session are placed contiguously in one queue. After all queues are serviced once, the AES Cipher will be processing 11 unique sessions with 11 different session keys at any one time (more sessions may be queued externally).

Using the Multi-Session Pipeline architecture, aggregate throughputs comparable to those achieved with the pipeline approach described in section 5.1.1 are possible. Note that while the throughput of a fully pipelined design was given as:

$$Throughput_{PIPE(K=11)} = \frac{128}{T_{ARCH}} = \frac{128}{T_{ROUND} + T_{OH}},$$

the aggregate throughput (across all sessions) of a Multi-Session Pipeline design is:

$$Throughput_{AGG(MS-PIPE)} = \frac{128}{T_{ARCH}} = \frac{128}{T_{ROUND} + T_{OH}},$$

and the throughput for any one of the CBC sessions is the same as for a pipeline with K=1:

$$Throughput_{CBC-Session(MS-PIPE)} = \frac{128}{NR * T_{ROUND} + NR * T_{OH}},$$

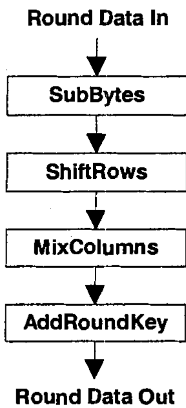
where NR is typically 11.

The Multi-Session Pipeline architecture provides enhanced aggregate throughputs and space savings over simply instantiating the basic architecture (pipeline with K=1) NR times.

5.2 Algorithmic Options

The basic round function of the AES cipher algorithm appears below in Figure 25. The only areas where optimisations can be achieved are in the SubBytes and MixColumns operations [12]. The ShiftRows operation is a permutation of bytes and requires no hardware to implement, while AddRoundKey consists solely of an XOR of one 128 bit word with another 128 bit word.

Figure 25 AES Cipher Round Algorithm



5.2.1.1 SubBytes Optimization

As stated in section 4.3.1.1, SubBytes may be implemented using either a look-up table, or by implementing the following equation.

$$\begin{bmatrix} b_0' \\ b_1' \\ b_2' \\ b_3' \\ b_4' \\ b_5' \\ b_6' \\ b_7' \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \\ b_4 \\ b_5 \\ b_6 \\ b_7 \end{bmatrix} + \begin{bmatrix} 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \\ 0 \end{bmatrix}$$

The look-up table approach offers a throughput advantage, but requires a larger area [13]. For the AES Cipher, 16 256x8 LUTs are required for SubBytes (assuming all 128 bits of the block are processed simultaneously. Several ways have been suggested for improving the throughput of the LUT approach, including using a twisted binary decision diagram, or going with a full custom approach and optimising the S-BOX at the transistor level.

Several authors [12], [13], [19] – [24] have proposed various methods to reduce the delay associated with the SubBytes process. The logic minimization and low fanout decoding approach discussed in [19] appears to offer the best combination of low delay and low gate/transistor count.

5.2.1.2 MixColumns Optimization

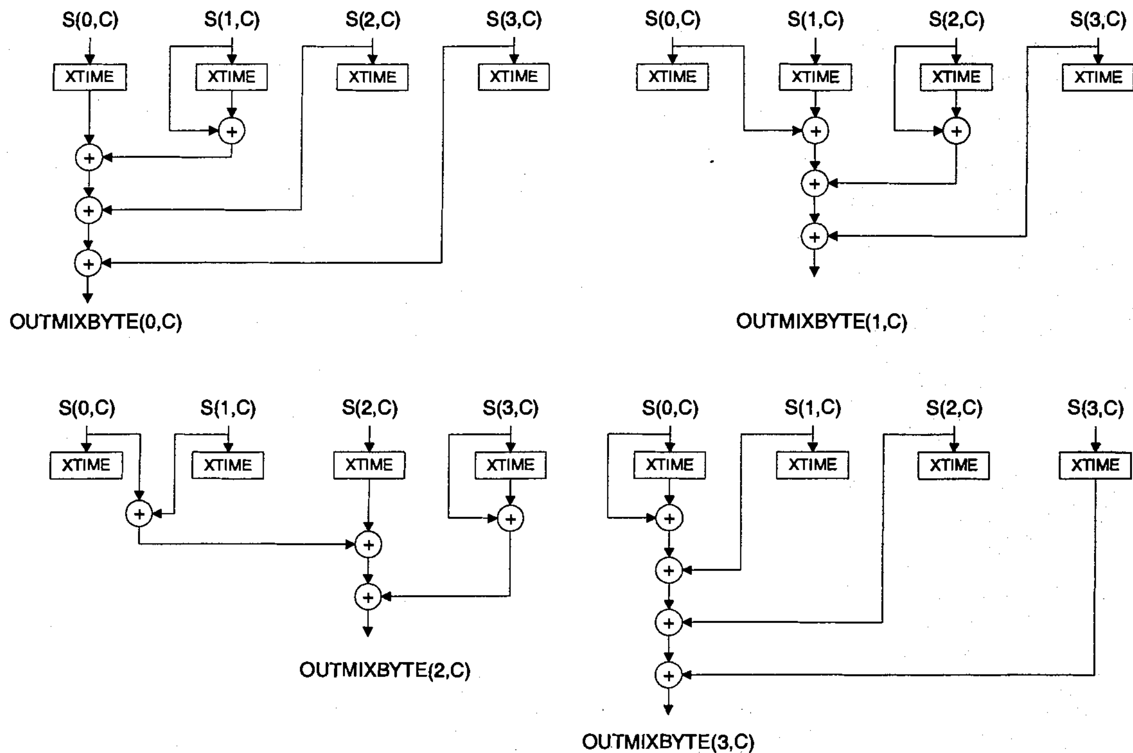
The MixColumns operation requires the implementation of the following equations for each column of the state:

$$S_{0,C}' = 2 * S_{0,C} + 3 * S_{1,C} + 1 * S_{2,C} + 1 * S_{3,C}$$
$$S_{1,C}' = 1 * S_{0,C} + 2 * S_{1,C} + 3 * S_{2,C} + 1 * S_{3,C}$$
$$S_{2,C}' = 1 * S_{0,C} + 1 * S_{1,C} + 2 * S_{2,C} + 3 * S_{3,C}$$
$$S_{3,C}' = 3 * S_{0,C} + 1 * S_{1,C} + 1 * S_{2,C} + 2 * S_{3,C}$$

If one were to implement these equations directly into VHDL without care, the synthesizer may produce the logic diagrams shown in Figure 26. Note that XTIME left shifts the input byte by 1 position then XORs the result with 00011011 (0x1B) if the MSB of the original byte was 1.

By analysing the delays incurred for each of the output bytes, it can be seen that OutMixByte(0,C) requires up to 5 gate delays to be processed. However, OutMixByte(2,C) only requires a maximum of 3 gate delays. Therefore, this implementation is not optimised.

Figure 26 Unbalanced MixColumns implementation



By re-ordering the terms of the input equations to the following, a balanced implementation will be created that results in all output bytes experiencing a maximum of 3 gate delays.

$$S_{0,C}' = 1 * S_{2,C} + 1 * S_{3,C} + 2 * S_{0,C} + 3 * S_{1,C}$$
$$S_{1,C}' = 1 * S_{0,C} + 1 * S_{3,C} + 2 * S_{1,C} + 3 * S_{2,C}$$
$$S_{2,C}' = 1 * S_{0,C} + 1 * S_{1,C} + 2 * S_{2,C} + 3 * S_{3,C}$$
$$S_{3,C}' = 1 * S_{1,C} + 1 * S_{2,C} + 2 * S_{3,C} + 3 * S_{0,C}$$

5.2.1.3 T-BOX Implementation

An alternative to the use of the traditional SubBytes and MixColumns implementations is that of the T-BOX [12]. A T-BOX is a look up table approach that not only incorporates SubBytes, but ShiftRows and MixColumns as well. Algebraically, the T-BOX can be expressed as follows [12]:

$$\begin{bmatrix} S'_{0,C} \\ S'_{1,C} \\ S'_{2,C} \\ S'_{3,C} \end{bmatrix} = \begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix} \begin{bmatrix} SubBytes(S_{0,C}) \\ SubBytes(S_{1,C+1}) \\ SubBytes(S_{2,C+2}) \\ SubBytes(S_{3,C+3}) \end{bmatrix}$$

This matrix can be expressed by the following four equations:

$S_{0,C}' = 2*SubBytes(S_{0,C}) + 3*SubBytes(S_{1,C+1}) + 1*SubBytes(S_{2,C+2}) + 1*SubBytes(S_{3,C+3})$
$S_{1,C}' = 1*SubBytes(S_{0,C}) + 2*SubBytes(S_{1,C+1}) + 3*SubBytes(S_{2,C+2}) + 1*SubBytes(S_{3,C+3})$
$S_{2,C}' = 1*SubBytes(S_{0,C}) + 1*SubBytes(S_{1,C+1}) + 2*SubBytes(S_{2,C+2}) + 3*SubBytes(S_{3,C+3})$
$S_{3,C}' = 3*SubBytes(S_{0,C}) + 1*SubBytes(S_{1,C+1}) + 1*SubBytes(S_{2,C+2}) + 2*SubBytes(S_{3,C+3})$

Equations for all 16 bytes of the state can be generated from this by replacing C with the column (0 to 3) being operated on. ShiftRows is implemented in these equations by adding 0, 1, 2, or 3 to the column value C.

A look-up table holding 1x, 2x, and optionally 3x the SubBytes value should be incorporated. Alternatively, additional ROMs may be used to hold the 2x and 3x values. When fully implemented, up to 48 256x8 ROMs (three for each byte of the state) may be required for one round of the algorithm when using the T-BOX approach. The T-BOX approach is one of the methods chosen in this Thesis for the "space-optimised" AES cipher.

The T-BOX approach is meant to reduce or eliminate the following delays associated with the standard implementation:

- Delay in generating 2x the SubByte value
- Delay associated with the multiple levels of XORing required in the MixColumns procedure

Gate delay is reduced at the cost of increased gate count and net delay, and therefore area. Whereas the standard AES Cipher implementation requires 16 ROMs or look-up tables, the T-BOX approach can require up to 48 for each round.

Note that the T-BOX approach is only useful if the savings in gate delay exceed the increases in net delay.

5.3 Implementation Options

Section 5.1 discussed how, given a value for T_{ROUND}, the design architecture affects the delay of the design. Section 5.2 showed how different algorithmic design choices can affect T_{ROUND}. T_{ROUND} is also affected by the choice of physical implementation. Among the choices are FPGA, ASIC, or full-custom ASIC, all of which offer advantages over the others. The decision to pursue one option over the others is often driven by one or more of the characteristics listed in Table 3.

Table 3 Key Physical Implementation Characteristics

Characteristic	FPGA	Standard Cell ASIC	Custom ASIC
Initial Time to Market	Fastest	Medium	Slowest
Development Cost	Lowest	Medium	Highest
Tooling Costs	Lowest	Medium	Highest
Device Cost	Highest	Lowest	Medium
Throughput	Lowest	Medium	Fastest

An FPGA development offers the fastest initial time to market, and lowest development and tooling costs by saving on physical design, layout, and tape-out expenses

associated with ASIC approaches (note that the time to market for a production ready design is equivalent across all options). However, device costs associated with FPGA-based designs are high. As an example, a design that may incur a device cost (silicon + packaging) of $10-$15 may require a $50+ FPGA. A hard-copy FPGA or structured ASIC program that will lower the device cost to near ASIC levels could be considered. However, this is at the expense of increased development costs and schedule impact (structured ASIC programs have costs and schedule impacts similar to those of traditional chip developments). In addition, the FPGA design may not satisfy the throughput requirements of the target application.

Another consideration is the anticipated volumes. If the volumes are low, an FPGA design (despite the higher per device costs) will offer a lower program cost (development cost + volume * device costs). However, as volumes increase to a certain level, FPGA and ASIC program costs will crossover such that the FPGA approach is more expensive.

Finally, ASIC and in particular full-custom ASIC approaches can achieve higher throughputs than in FPGA-based designs. As an example, most FPGA based designs in the literature achieve throughputs in the hundreds of Mbps (in CBC mode) and up to 20 Gbps or more in non-feedback mode [20] – [24], while ASIC approaches have achieved those rates and greater.

CHAPTER 6 DETAILED DESIGN OF AES

The AES implementations discussed in this Thesis were designed and verified using VHDL and Xilinx Synthesis Tools.

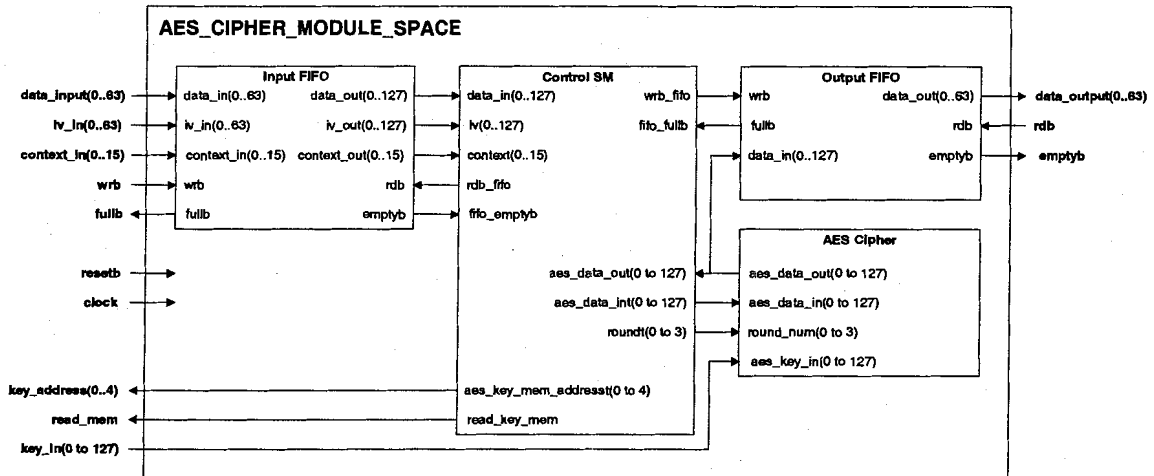
Two different AES implementations have been developed for this thesis. The first implementation, based on the basic or pipeline architecture with K=1 (as shown in Figure 22) is optimised for space. The second, based on the Multi-Session Pipelined architecture of Figure 24 is optimised for aggregate throughput.

The following sections describe the detailed design of the AES Cipher for these two implementations.

6.1 Space Optimised AES Cipher

Figure 27 shows a block diagram of the basic AES Cipher Module. As can be seen, it is implemented using four sub-modules: Input FIFO, Control State Machine, Output FIFO and importantly, a single AES Cipher sub-module. The AES_CIPHER_MODULE_SPACE block implements the top level connections between each sub-module, as well as the input/output interface to the testbench environment.

Figure 27 AES Cipher Module Block Diagram



The Input FIFO sub-module buffers the write transactions from the testbench and converts the data width from 64 bits on the testbench to the 128 bit data path required by the encryption engine. The Input FIFO is also used to buffer the context and initialisation vectors. Similarly, the Output FIFO sub-module buffers the encryption results from the encryption engine and converts the 128 bit data path utilized internally to a 64 bit data path expected by the testbench.

The Control State Machine sub-module controls most operations of the encryption engine including the reading of data to be encrypted from the Input FIFO and the writing of encrypted data to the Output FIFO. The sub-module implements a state machine that governs the operation of the encryption engine for each clock cycle of the encryption process.

The AES Cipher sub-module implements the actual AES algorithm itself (with the exception of the key generation logic).

The following table describes the operation of each of the input and output signals on the top level design.

Table 4 Pin Description of Space Optimised AES Cipher Module

Signal Name	Input/Output	Description
data_input(0 to 63)	Input	Supplies the data input to the Cipher module for encrypting. One half of the 128 bit AES block is provided on each valid clock cycle.
iv_in(0 to 63)	Input	Supplies the IV input to the Cipher module for encrypting data in CBC mode. One half of the 128 bit IV is provided on each valid clock cycle.
context_in(0 to 15)	Input	Provides the Cipher module with knowledge as to how it should process the associated data. The encoding of context_in is as follows: Bit 0: '1' = Start of Packet. '0' = middle or end of packet. Bit 1: "1' = Encryption, '0' = Decryption Bits 2:3: Indicate the mode the cipher is to operate in. "01" = ECB mode. "10" = CBC mode. All other values are ignored. Bits 4:15: Indicate the key index to be used.
wrb	Input	Indicates the data on data_input, iv_in, and context_in should be written into the Input FIFO.
fullb	Output	Indicates the status of the input FIFO. '0' indicates that the Input FIFO is full, and '1'indicates that the Input FIFO has room for at least one more 128 bit transaction.
key_address(0 to 4)	Output	Provides the key address of the next required key.
read_mem	Output	Provides a read strobe for the associated key memory.
key_in(0 to 127)	Input	The key to be used in the encryption process for each round.
data_output(0 to 63)	Output	The result of encrypting the input data with the context information as directed in contex_in, and the associated key.
rdb	Input	Indicates that the testbench is ready to accept new data from the output FIFO.
emptyb	Output	Indicates the status of the output FIFO. '0' indicates

Signal Name	Input/Output	Description
		that the output FIFO is empty (and therefore no need to continue asking), and '1'indicates that the Output FIFO has at least one more 128 bit transaction before going emptyb.
clock	Input	Provides a synchronous signal to all the clocked elements in the design. Clock is active on the rising edge.
resetb	Input	Provides a synchronous reset to all of the clocked element s in the design.

6.1.1 Input FIFO Sub-module

The Input FIFO sub-module implements three circular buffers of 8 locations each. The number of locations is configurable depending on the access speeds of the testbench and the encryption rate of the engine itself. Two of the buffers feature 64 bit wide locations, while the third buffer uses 16 bit wide locations. All buffers share the same read and write pointer to ensure they are synchronized.

Data to be encrypted is written into the FIFO in 64 bit transactions. Therefore, for AES applications, two transactions must occur to write the complete 128 bit AES block into the FIFO. At the same time, the IV and context information must also be loaded into the FIFO. Even if ECB mode is being used, the IV field as well as any unused bit locations in the context field, should be set to 0.

Upon each write from the testbench, the write pointer is incremented one position, and an internal contents counter is incremented. To protect the FIFO from overrun conditions, the write pointer is compared with the read pointer. If the write pointer is within 2 locations of the read pointer, the Input FIFO will assert the fullb signal to the testbench. The testbench should not attempt to write new data into the FIFO until the fullb signal is de-asserted.

The Control SM block controls reading from the FIFO. Upon each read from the Control SM block, 2 locations are read from each buffer and concatenated together to form the 128 bit or 32 bit word required by the Control SM sub-module. The read pointer is incremented by two and compared to the write pointer. If the new read pointer and write pointer are equal to each other, the FIFO is empty and the emptyb signal is asserted to the Control SM. To avoid FIFO under-runs, the Control SM must not attempt to read from the FIFO when emptyb is asserted.

6.1.2 Output FIFO Sub-Module

The Output FIFO sub-module implements one circular buffer of 8 locations. The number of locations is configurable depending on the access speeds of the testbench and the encryption rate of the engine itself. The buffer features 64 bit wide locations.

Encrypted data is presented to the FIFO as a 128 bit word. On each write transaction initiated by the Control SM, two locations are filled. The first location corresponds to bits 0 to 63 of the encrypted data block, and the second location corresponds to bits 64 to 127. Upon each write from the Control SM, the write pointer is incremented two positions, and an internal contents counter is incremented. To protect the FIFO from overrun conditions, the write pointer is compared with the read pointer. If the write pointer is equal to the read pointer, and the contents counter indicates the FIFO is holding at least 6 units of data, the Output FIFO will assert the fullb signal to the Control SM. The Control SM should not attempt to write new data into the FIFO until the fullb signal is de-asserted. Therefore, the Control SM must go into a holding state until the Output FIFO is no longer full.

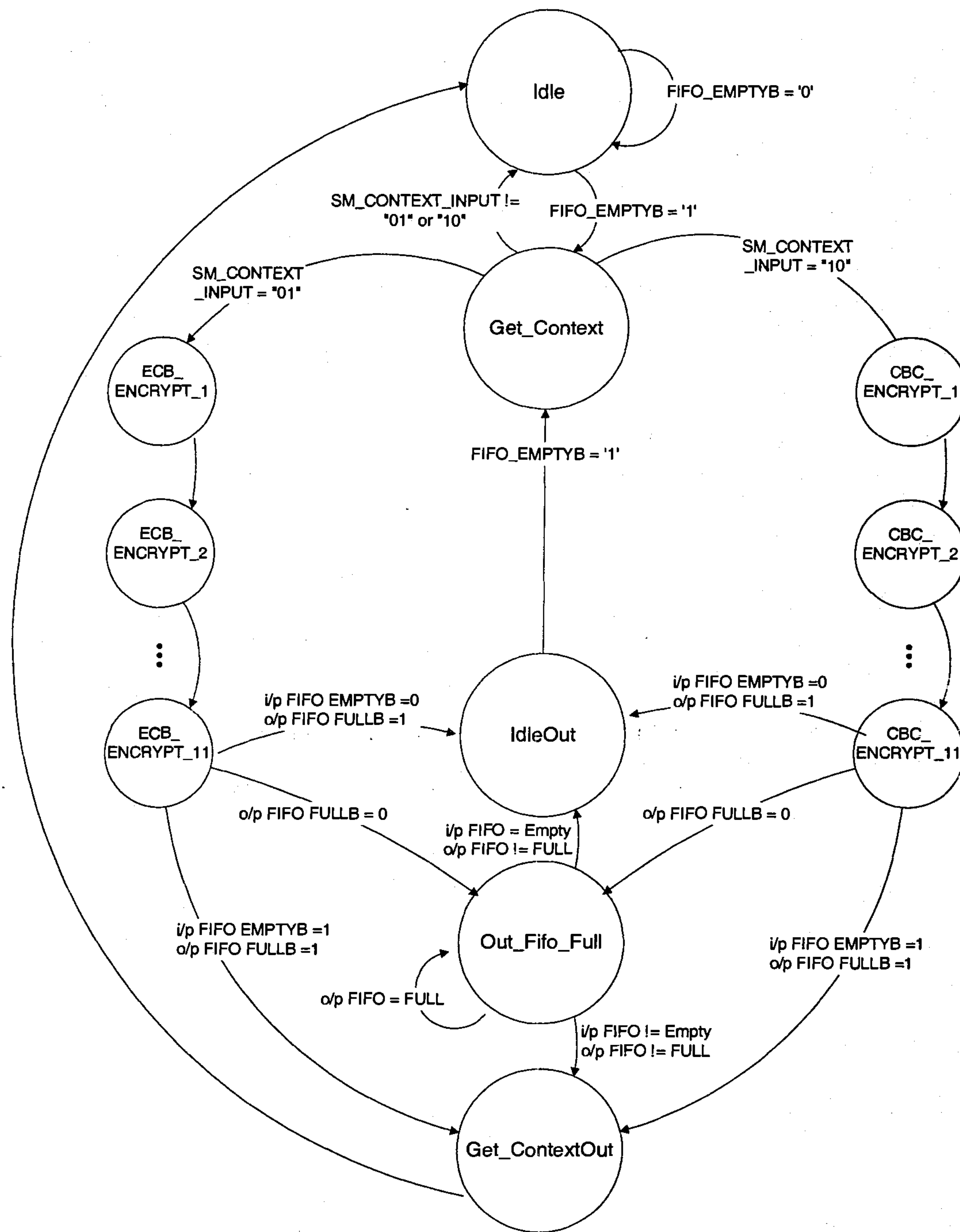
The testbench controls reading from the Output FIFO. Upon each read from the testbench, 1 location is read from the buffer. Therefore, for AES applications, two transactions must occur to read the complete 128 bit AES block from the Output FIFO. The read pointer is incremented by one and compared to the write pointer. If the new read pointer is within one location of the write pointer, the FIFO is empty and the emptyb signal is asserted to the testbench. To avoid FIFO under-runs, the testbench must not attempt to read from the FIFO when emptyb is asserted.

6.1.3 Control SM Sub-module

The Control SM (State Machine) sub-module implements a state machine of 27 states that controls the operation and sequencing of the cipher. Figure 28 is the state diagram for the Control SM. The operation is as follows:

- Initially, the state machine is in the IDLE state and remains so until the FIFO_EMPTYB signal from the input FIFO block is '1', indicating that data is in the FIFO. Once FIFO_EMPTYB is '1', the state machine transitions to the Get_Context state.
- In the Get_Context state, the state machine reads the context bits to determine how processing should proceed. If bits 2 and 3 of the context input are equal to "01", the state machine transitions to the ECB_ENCRYPT_1 state. If bits 2 and 3 are equal to "10", the state machine transitions to the CBC_ENCRYPT_1 state.

Figure 28 Control SM State Diagram



o In the ECB_ENCRYPT_1 state, the state machine drives the data to be encrypted to the AES Cipher sub_module. On subsequent clock cycles, the state machine cycles through states 2 through 10. In states 2 through 10, the

controller simply feeds back the output of the previous round to the input of the next round.

- In the ECB_ENCRYPT_11 state, the data is completely encrypted having traversed through all rounds of the state. In this state, the controller examines the state of the FIFO_EMPTYB flag from the Input FIFO and the FIFO_FULLB flag from the Output FIFO. If the FIFO_FULLB flag is '0', the state machine transitions to the Out_FIFO_Full state. Otherwise if both the FIFO_EMPTYB flag and FIFO_FULLB flag are '1', the state machine transitions to the Get_ContextOut state because there is new data available to encrypt and the output FIFO is empty. If the FIFO_EMPTYB flag is '0' and the FIFO_FULLB flag is '1', the state machine transitions to the IDLEOUT state due to the fact that there is no more data to encrypt.

- The functioning of the CBC_ENCRYPT_1 through CBC_ENCRYPT_11 states is similar to that of the ECB_ENCRYPT_X states. The primary difference is that in CBC_ENCRYPT_1, the input data to be encrypted must be XOR'ed with the IV if this is the first block of the packet. If not the first block of the packet, the input data to be encrypted must be XOR'ed with the previous encryption result.

- In the Get_ContextOut state, the state machine retrieves the next data, IV and context information required to encrypt the next block of data.

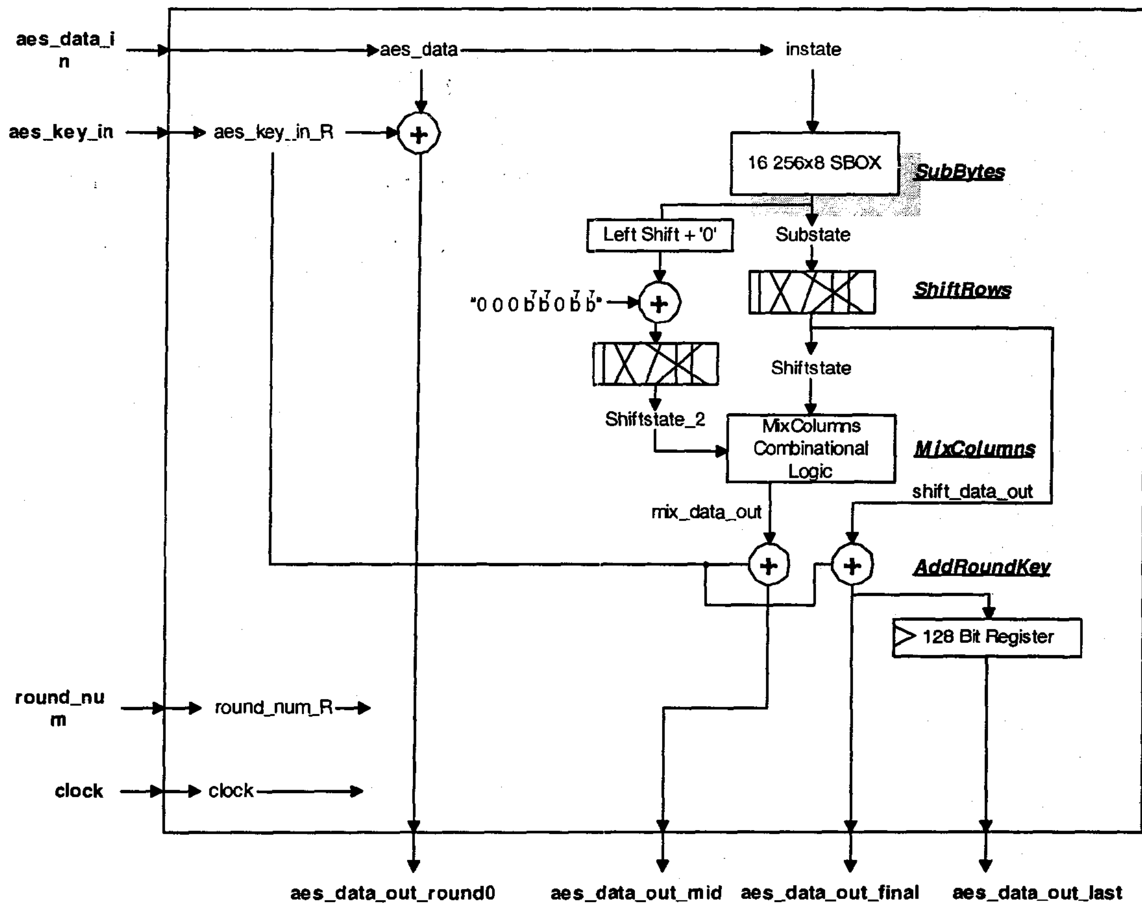
- In the Out_FIFO_Full state, the state machine monitors the setting of the FIFO_FULLB flag to determine when space is available in the Output FIFO. When space becomes available, the encrypted data is written into the FIFO, and the state machine transitions to the IDLEOUT or Get_ContextOut state depending on whether or not data in the Input FIFO is waiting to be encrypted.

- In the Get_ContextOut state, the state machine writes the encrypted data into the Output FIFO and reads the data, IV and context for the next block of data to be encrypted.

6.1.4 AES Cipher Sub-module

Two versions of the AES Cipher module have been developed. One version implements the traditional S-Box approach for SubBytes. A simplified block diagram of this version of the AES Cipher sub-module is shown in Figure 29.

Figure 29 AES Cipher Sub-module Block Diagram



Input data to the cipher sub-module is placed into an array named Instate which mimics the state construct in the AES specification [2]. All 16 byte values of the state are then used as lookup addresses for 16 ROMs, the output of which is named Substate. The Substate signal then takes two paths. One path implements the Shiftrows function only, and results in a new signal named Shiftstate. The other path left shifts the Substate signal one position and XORs the result with "000$B^0B^0$0B^0B^0" where B^0 is the most significant bit of the Substate signal (prior to left shifting). If B^0 is 0, then the XOR function has no effect. If B^0 is 1, the left shifted value is then XOR'd with "00011011" or 0x1B. This process creates a new signal termed Shiftstate_2.

Shiftstate and Shiftstate_2 are used as part of the MixColumns combinatorial logic. The MixColumns logic implements the following balanced MixColumns equations:

$$S_{0,C}' = 1 * S_{2,C} + 1 * S_{3,C} + 2 * S_{0,C} + 3 * S_{1,C}$$
$$S_{1,C}' = 1 * S_{0,C} + 1 * S_{3,C} + 2 * S_{1,C} + 3 * S_{2,C}$$
$$S_{2,C}' = 1 * S_{0,C} + 1 * S_{1,C} + 2 * S_{2,C} + 3 * S_{3,C}$$
$$S_{3,C}' = 1 * S_{1,C} + 1 * S_{2,C} + 2 * S_{3,C} + 3 * S_{0,C}$$

$1*S_{r,c}$ represents the various Shiftstate values, $2*S_{r,c}$ represents the Shiftstate_2 values, and $3* S_{r,c}$ is the result of XORing Shiftstate and Shiftstate_2 for the appropriate row and column values.

The output of the MixColumns operations is XOR'ed with the AES_KEY_IN value for this particular round as part of the AddRoundKey function to create the AES Cipher output for rounds 1 through 9 of the Cipher.

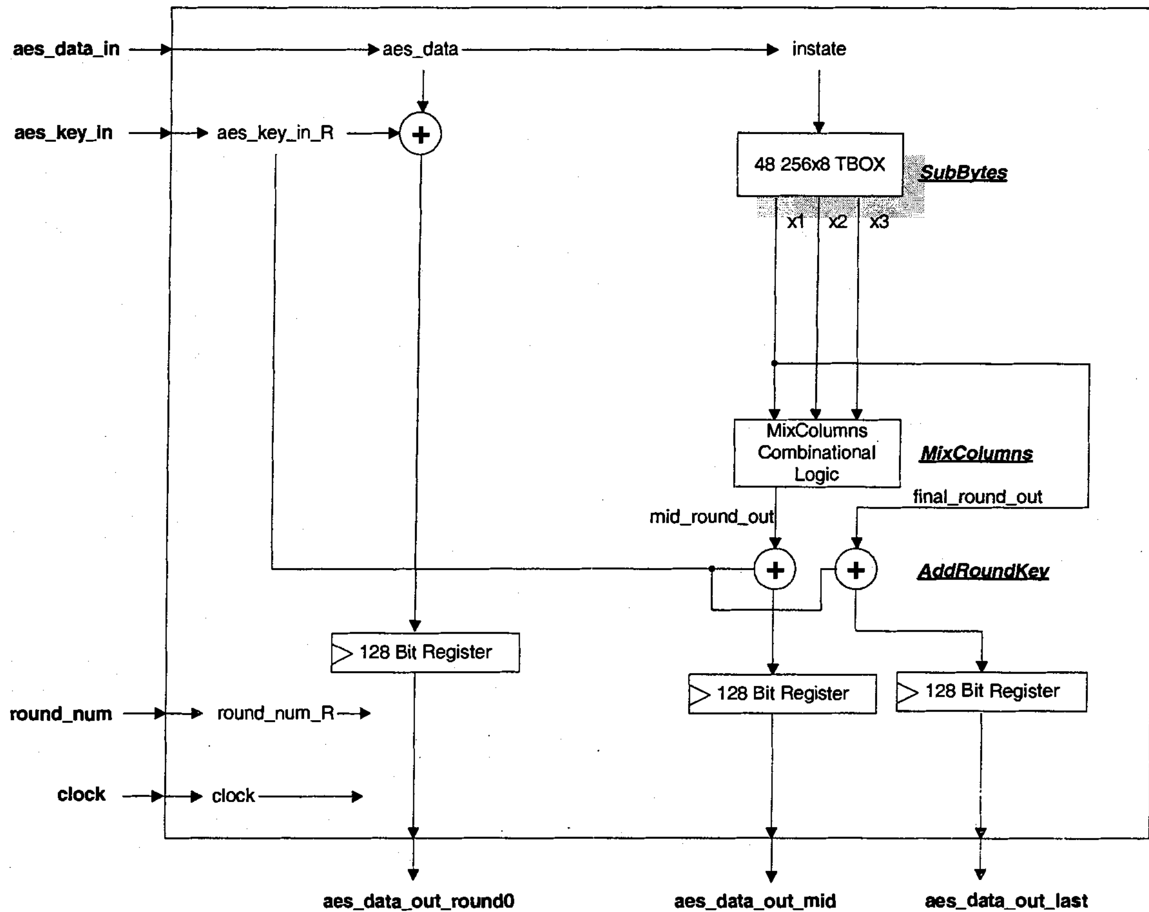
Note that the AES Cipher module provides four possible outputs:

- One output (AES_DATA_OUT_ROUND0) computes the XOR of the key and input (for round 0)

- One output (AES_DATA_OUT_FINAL) computes the XOR of the key and the result of SubBytes (for round 10)
- One output (AES_DATA_OUT_MID) computes the XOR of the key and the result of the MixColumns operation (for rounds 1 through 9)
- One output (AES_DATA_OUT_LAST) provides a registered version of AES_DATA_OUT_FINAL

An alternative AES Cipher module was developed that utilized the T-BOX approach described in Section 5.2.1.3 and [12]. Figure 30 shows the block diagram of this alternate version of the AES Cipher module.

Figure 30 AES Cipher Module Using T-Box Approach

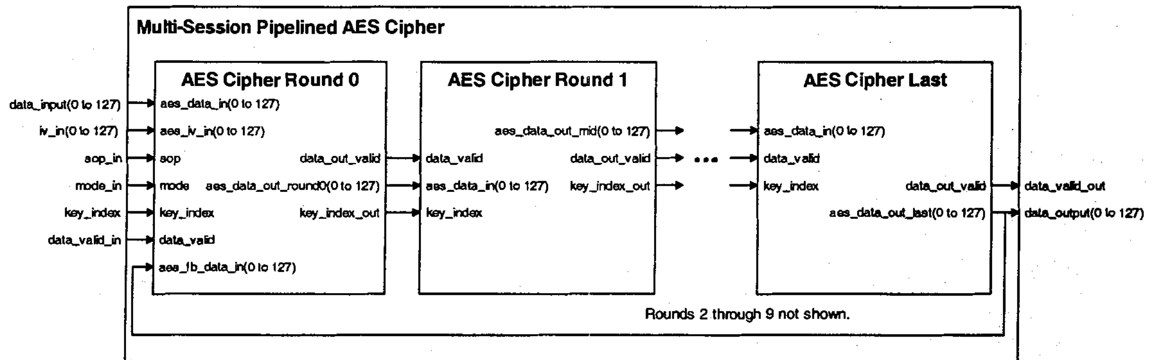


6.2 Multi-Session Pipelined AES Cipher

The following sections describe the implementation of the Multi-Session Pipelined AES Cipher module, also referred to as the throughput optimised design.

Figure 31 shows a block diagram of the Multi-Session Pipelined AES Cipher Module.

Figure 31 Multi-Session Pipelined AES Cipher Module



There are several high level differences between the design of Figure 31, and that of section 6.1. These include:

- The top-level interface is changed to support 128 bit wide data paths for the input data, output data, and IV signals. As well, the Key Interface is removed. Finally, signals are added to pass mode, key index, start of packet, and data valid information to the Cipher Module.
- The Input FIFO, Output FIFO and Control SM are removed in order to support a new 128 bit block of data on every clock cycle (for maximum throughput).
- Finally, a separate AES Cipher round sub-module is instantiated for each round of the design.

The individual AES Cipher round sub-modules that form the basis of the design store the pre-computed individual round-keys required for their particular position in the pipeline. For instance, the round 0 sub-module will only contain round 0 round-keys for the sessions in use. Likewise, the round 5 sub-module will only contain round 5 round-keys. The round-key to use will be selected based on the key_index signal. Since the AddRoundKey function of the AES algorithm is always the last operation to be

performed in a round, the key memory will be able to provide the correct session key by the time it is needed. The value of the key_index signal will propagate with the data through each sub-module of the design to ensure that the correct key is used on a per round basis to encrypt the data.

Since the Multi-Session Pipelined AES Cipher will produce a new 128 bit encryption result on every clock cycle, the data_valid_in signal is provided to qualify the validity of the input data. When the data on the data_input signal is valid, the data_valid_in signal will be high. This will propagate through each stage of the pipeline and will inform the downstream processing block that the output_data signal is valid.

The pin description of the Multi-Session Pipeline AES Cipher is presented in Table 5.

6.2.1 AES Cipher Sub-module

The Multi-Session Pipelined AES Cipher Module utilizes modified versions of the Cipher Module shown in Figure 29. For rounds 1 through 10 of the design, the primary differences are that the sub-module contains the pre-computed round-keys for each of the supported sessions. In addition, the sub-modules register and pass the key_index and data_valid signals as they propagate with the data they pertain to.

The AES_Cipher_Round0 sub-module is further modified to check the status of the mode and SOP signals. If the mode signal '0', the data is to be encrypted using ECB mode, and is simply XOR'ed with the round-key. If the mode signal is '1', the data is to be encrypted using CBC mode. In this case, the sub-module also checks the value of SOP. If SOP is '1', the block of data corresponds to the start of packet, and the data is XOR'ed with the IV signal before being XOR'ed with the round-key. If SOP is '0', the

data is XOR'ed with the result of encrypting the last block of data on this session before being XOR'ed with the round-key.

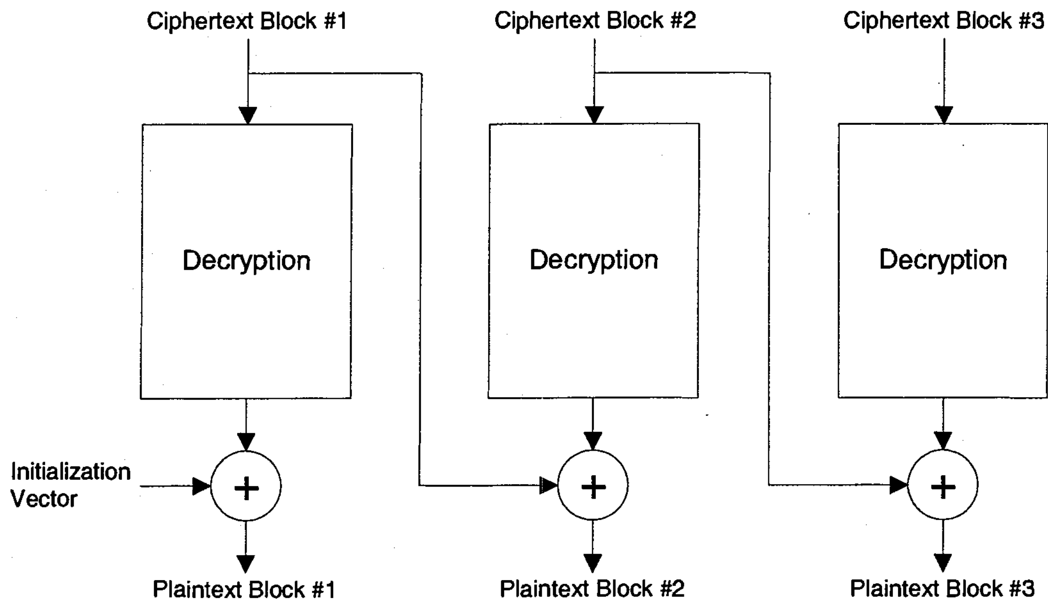
Table 5 Pin Description of the Multi-Session Pipeline AES Cipher Module

Signal Name	Input/Output	Description
data_input(0 to 127)	Input	Supplies the data input to the Cipher module for encrypting.
iv_in(0 to 127)	Input	Supplies the IV input to the Cipher module for encrypting data in CBC mode.
key_index	Input	Provides the Cipher module with knowledge as to which round-key to use. May be expanded as needed to support the required number of sessions.
mode_in	Input	Indicates whether the block should be encrypted using ECB or CBC mode. '0' = ECB Mode. '1' = CBC Mode.
sop_in	Input	Indicates whether the block represents the start of a packet or not. '1' = Start of Packet. '0' = middle or end of packet.
data_valid_in	Input	Indicates that the data presented on the data_input, iv_in, key_index, mode_in, and sop_in signals are valid.
data_valid_out	Output	Indicates that the data presented on the data_output signal is valid.
data_output(0 to 127)	Output	Output data for the Multi-Session Pipelined AES Cipher.
clock	Input	Provides a synchronous signal to all the clocked elements in the design. Clock is active on the rising edge.
resetb	Input	Provides a synchronous reset to all of the clocked elements in the design.

6.3 Inverse Cipher Design

One of the goals of this Thesis was to create a modular design that with minor alteration, could be re-used for the Inverse Cipher and Key Generation functions. In light of this, the Input FIFO and Output FIFO are 100% re-used for the Inverse Cipher. Since keys are used in reverse order with the Inverse Cipher, the Control SM module is altered to decrement key address and round values. Finally, the Cipher sub-module is necessarily updated to implement the actual Inverse Cipher algorithm as described in Section 4.3.2. The design of the inverse cipher sub-module is similar to that of the cipher module depicted in Figure 29 with the exception that an extra register and multiplexer is required to support CBC mode. The register is used to hold the IV value (for the first block of data) or the prior block of input data. Figure 32 depicts the general configuration of an Inverse Cipher in CBC mode.

Figure 32 AES Inverse Cipher in CBC Mode



6.4 Key Expansion Design

The only variation in the design of the Key Expansion module from the AES cipher module is the Key Expansion sub-module replaces the AES cipher sub-module. The Input FIFO, Output FIFO and Control SM are 100% re-used from the AES Cipher design.

CHAPTER 7 AES DESIGN VERIFICATION

The AES Cipher was designed using Xilinx Synthesis Tools, and verified using the Modelsim verification environment. Before discussing test results for the different implementations, the verification strategy is introduced.

AES design verification is composed of two components:

- Design of the Testbench
- AES Cipher Module Verification

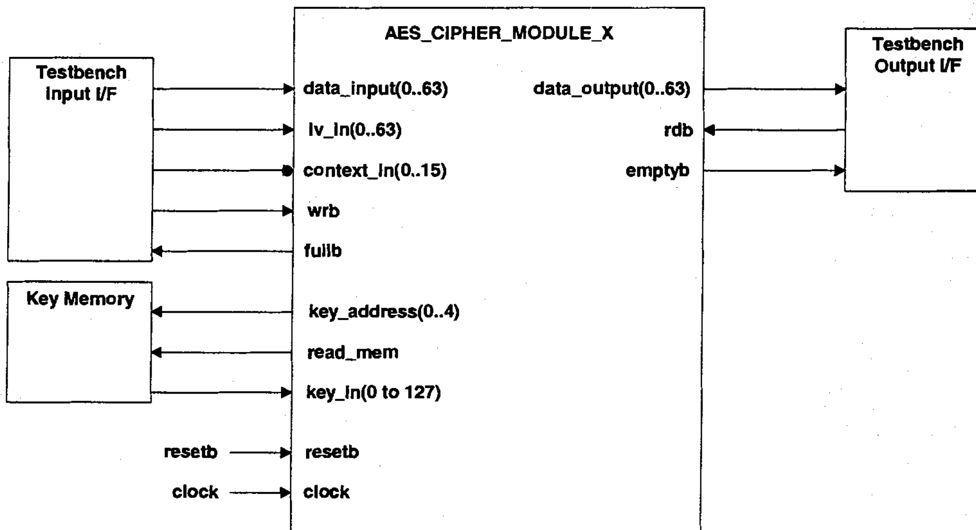
The following sections describe these components in further detail.

7.1 Space-Optimised AES Testbench Design

Figure 33 on the next page shows the connections from the testbench to the space-optimised AES Cipher Module. The testbench performs three general operations:

1. Operation of the input interface
2. Operation of the output interface
3. Operation of the Key Memory interface

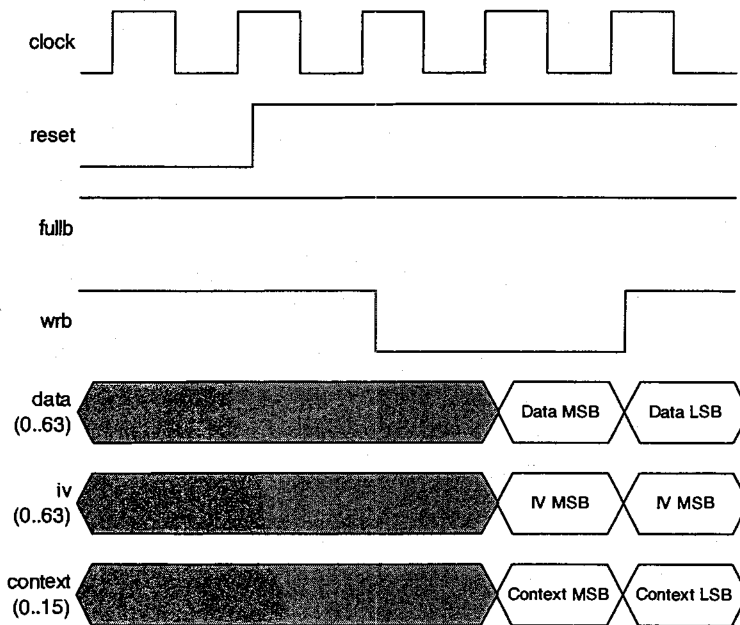
Figure 33 Testbench Connections



7.1.1 Input Interface

The input interface process controls the sequencing of data, IV, and context information to the AES Cipher Module. The input interface initially drives the reset signal to the design. Once out of reset, the testbench begins monitoring the fullb signal which indicates whether or not the Input FIFO in the AES Cipher Module has room to accept data. Transfer of data (including IV and context) from the testbench to the Cipher Module requires three clock cycles. On the first clock cycle (assuming fullb is high), the testbench will set the wrb signal to '0'. On the second clock cycle, the testbench will continue to assert wrb to '0', and will also drive the data and IV signals with the most significant 64 bits of the data and IV as well as the most significant 16 bits of the context information. On the third and final clock cycle, the testbench will de-assert wrb (to '1') and will drive the least significant bits of the data, IV and context signals. Note that if the testbench has data available to transfer to the Cipher Module, the testbench may not de-assert wrb on the third clock cycle. A timing diagram of this basic operation is presented in Figure 34.

Figure 34 Functional Timing Diagram of the Input Interface



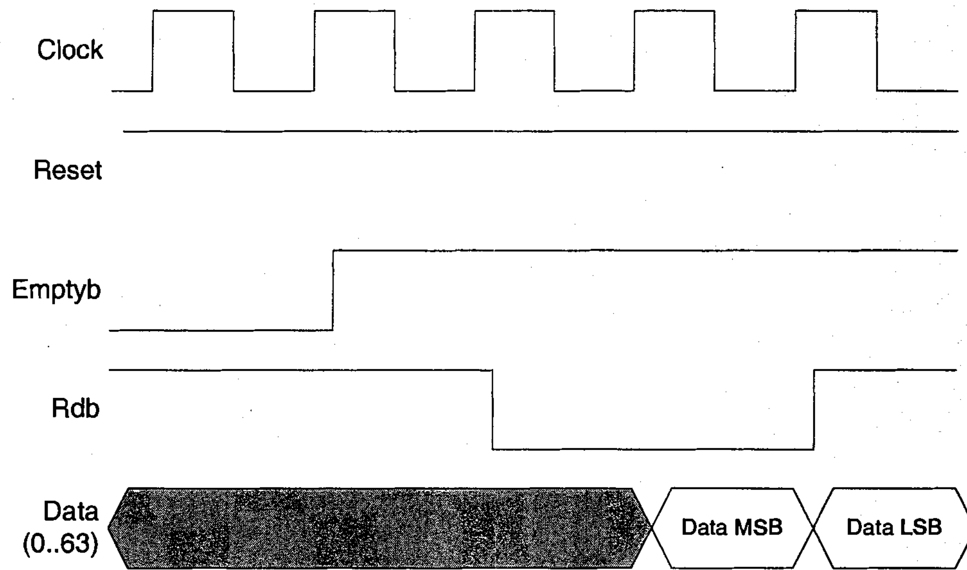
Note that if the fullb signal from the AES Cipher Module is asserted ('0'), the testbench will not drive the wrb signal to '0'. The data, IV, and context signals may be driven to any value.

7.1.2 Output Interface

The output interface controls the sequencing of data to be read from the AES Cipher Module. The testbench monitors the setting of the emptyb signal. When the Output FIFO of the AES Cipher Module is empty, emptyb will be '0', and the testbench will correspondingly de-assert rdb. Once the output FIFO contains data, the emptyb signal will be set to '1', at which time the testbench will assert the rdb signal to '0'. This will cause the AES Cipher Module to transfer encrypted data (in 64 bit segments) to the testbench.

The following figure depicts the functional timing on the Output Interface.

Figure 35 Output Interface Functional Timing

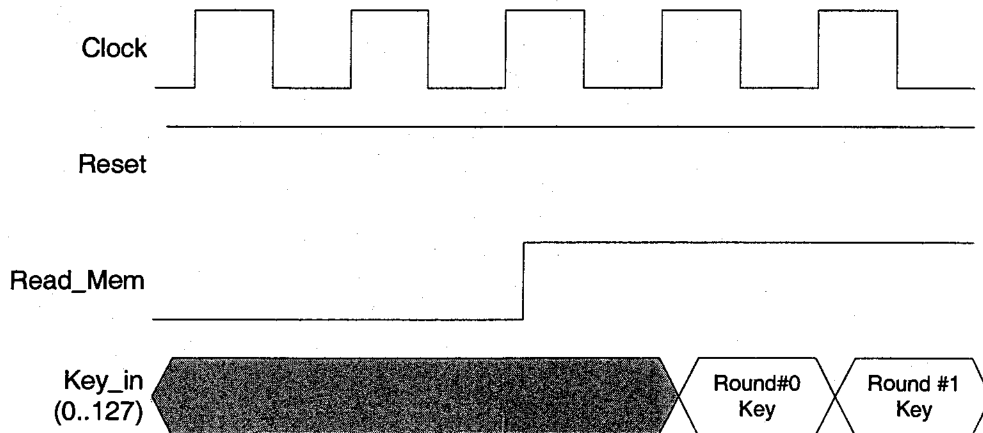


7.1.3 Key Interface

The testbench (used with the space optimised design) maintains a pre-computed Key Memory database. This database contains all of the round keys the Cipher is expected to use. The AES Cipher Module drives the read_mem signal into the testbench which, when set to '1', instructs the testbench to read a location from the key memory and send the key value read at that location back to the AES Cipher Module. The address into the Key Memory is formed by a concatenation of the key index and round number.

The following diagram depicts the functional timing on the Key Interface.

Figure 36 Key Interface Functional Timing



7.2 Multi-Session Pipelined AES Testbench Design

The testbench for the Multi-Session Pipelined AES Cipher is much simpler than that used for the space-optimised design. Instead of monitoring the status of the fullb and emptyb signals, the testbench now simply updates the data, IV, and associated context information on each rising clock edge. The testbench is designed to enforce an 11 clock cycle separation between CBC mode data blocks using the same session/key.

7.3 AES Cipher Module Verification

Once the design of the AES Cipher Module and associated testbench is complete, verification of the actual design can commence. The Xilinx FPGA design flow consists of four steps: Synthesis, Translate, Map, and Place and Route. Simulations may be run after each step, but for this Thesis, simulations were only run after the Synthesis and Place and Route steps. Simulations were run after the Synthesis step to catch syntax and logical errors while simulations were run after the Place and Route step to catch logical and timing errors in addition to determining the throughput of the design.

The AES Specification [11] contains test vectors that can be used to test the completed design to ensure that the expected results for a known input (data, IV, key) are obtained. In addition, RFC 3602 [14] contains test vectors for AES in CBC mode. Table 6 lists the test vectors utilized in this design as well as the expected and actual results.

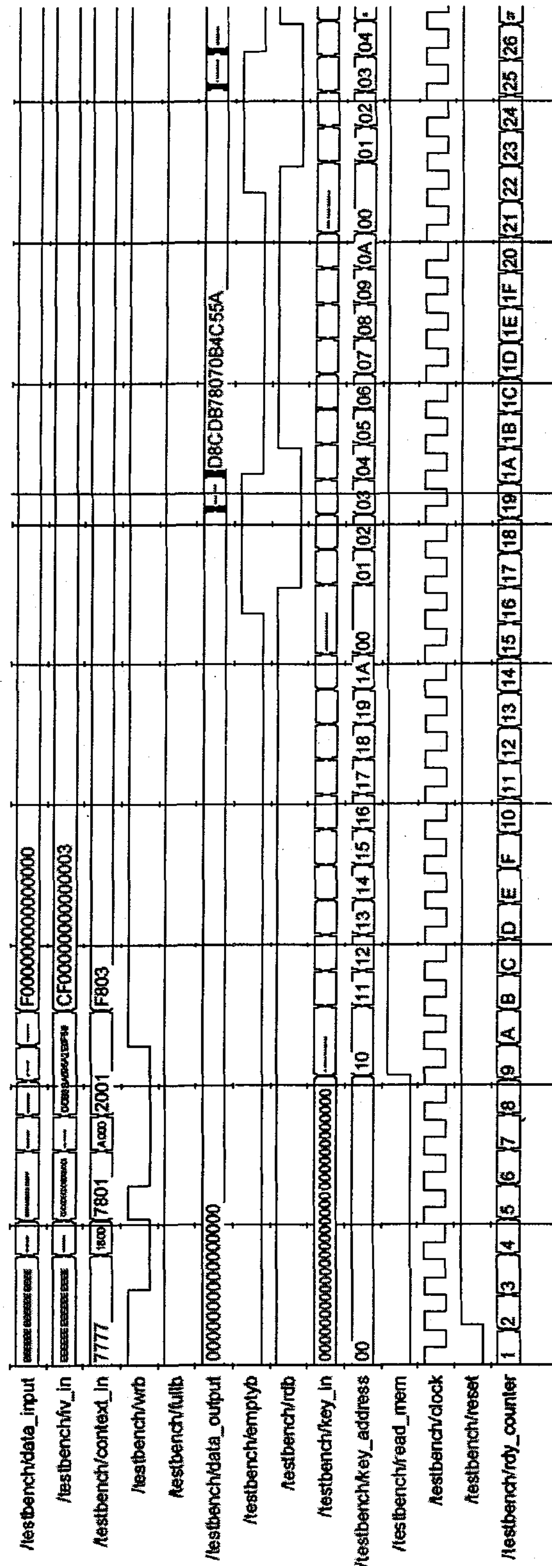
As can be seen, the AES Cipher design passes all test vectors. Note that additional test vectors can and should be run to ensure the design is system ready.

Table 6 Test Vectors used in the verification of AES

	Vector Set #1	**Vector Set #2**	**Vector Set #3**
Input Data	32 43 f6 a8 88 5a 30 8d 31 31 98 a2 e0 37 07 34	00 11 22 33 44 55 66 77 88 99 aa bb cc dd ee ff	00 01 02 03 04 05 06 07 08 09 0a 0b 0c 0d 0e 0f 10 11 12 13 14 15 16 17 18 19 1a 1b 1c 1d 1e 1f
Key	2b 7e 15 16 28 ae d2 a6 ab f7 15 88 09 cf 4f 3c	00 01 02 03 04 05 06 07 08 09 0a 0b 0c 0d 0e 0f	c2 86 69 6d 88 7c 9a a0 61 1b bb 3e 20 25 a4 5a
IV	Not applicable	Not Applicable	56 2e 17 99 6d 09 3d 28 dd b3 ba 69 5a 2e 6f 58
Mode	ECB	ECB	CBC (2 128 bit words)
Expected Result	39 02 dc 19 25 dc 11 6a 84 09 85 0b 1d fb 97 32	69 c4 e0 d8 6a 7b 04 30 d8 cd b7 80 70 b4 c5 5a	d2 96 cd 94 c2 cc cf 8a 3a 86 30 28 b5 e1 dc 0a 75 86 60 2d 25 3c ff f9 1b 82 66 be a6 d6 1a b1
Actual Result	39 02 dc 19 25 dc 11 6a 84 09 85 0b 1d fb 97 32	69 c4 e0 d8 6a 7b 04 30 d8 cd b7 80 70 b4 c5 5a	d2 96 cd 94 c2 cc cf 8a 3a 86 30 28 b5 e1 dc 0a 75 86 60 2d 25 3c ff f9 1b 82 66 be a6 d6 1a b1

Figure 37 presents a waveform diagram produced as a result of simulating the AES Cipher Module design with vector sets 2 and 3. In addition to verifying the design produces the expected results, it also shows that the design is capable of supporting both ECB and CBC mode. Additional simulation results are presented in Appendix A.

Figure 37 AES Cipher Encryption



CHAPTER 8 AES RESULTS

The following sections discuss the results of the AES designs implemented in this Thesis. Both size and performance numbers are included for the space-optimised and Multi-Session Pipelined designs and compared against prior works. Initial design of the AES Cipher Module utilized the XC2v3000fg676-6 FPGA. Subsequent testing utilized other FPGAs in order to compare the results of this Thesis with other published implementations.

8.1 Space Optimised AES Design Results

A design summary of the space optimised AES design is presented in Table 7. Note that this design utilized 4016 Xilinx FPGA slices with an equivalent gate count of 79K gates. During the design process, the AES Cipher Sub-module was found to be the limiting factor from a performance perspective. The extra overhead of the Input FIFO, Output FIFO, and Control SM is used to sequence data transfers to/from the design and provides a common interface to the testbench, but does not directly implement the AES algorithm. Therefore, for comparison purposes, the size characteristics of the AES Cipher Sub-module is also included in Table 7.

Table 7 Space Optimised AES Design Summary

Parameter	Complete AES Cipher	AES Cipher Sub-module
256x8-bit ROM	16	16
Number of Slices	4016	1454
Equivalent Gate Count	78,957	N/A

The following table (Table 8) details the performance characteristics of the space optimised AES design and compares the results with 3 other published works [15], [16], and [18]. An attempt was made to ensure that the works being compared also implement CBC mode. As can be seen, this design features a higher throughput than the other references. However, this comes at a cost of increased FPGA slices. The FPGA slices of the AES Cipher sub-module is also shown since it is not clear from the published results whether the other authors include overhead (such as the Input FIFO in this design) that is not directly related to implementing the AES algorithm.

An additional parameter "throughput (in Mbps)/Slice" is added in order to judge the relative efficiencies of the various designs. As can be seen, the design described in this Thesis offers the best efficiency. If only the slices in the AES Cipher module are included in the efficiency calculation, the design in this Thesis offers a significant improvement in efficiency over all other references in Table 8.

Table 8 Performance Characteristics of the Space Optimised AES Design

	This Design	Reference [15]	Reference [16]	Reference [18]
FPGA Type	XCV1000EFG 860-8	XCV1000 bg560-4	XCV1000 bg560-6	XCV600E-8BG432
FPGA Slices	4016 (1454)	5302	2902	4681
Clocks/Block	12	6	10	Not Published
Cipher Mode	ECB or CBC	CBC	ECB or CBC	All
Max. Clock Frequency	59.70 MHz	14.1 MHz	25.9 MHz	Not Published
Throughput	636.82 Mbps	300.1 Mbps	331.5 Mbps	310 Mbps
Throughput/ Slice	0.159 (0.438)	0.057	0.114	0.066

To eliminate the impact of different FPGAs on the test results, the AES Cipher Module was re-simulated with the Xilinx XCV1000bg560-6 FPGA. The results are listed in Table 9. As can be seen, the design described in this Thesis still offers higher throughput and greater efficiencies than the cited references.

Table 9 Performance Characteristics with Same FPGA

	This Design	Reference [15]	Reference [16]
FPGA Type	XCV1000 bg560-6	XCV1000 bg560-4	XCV1000 bg560-6
FPGA Slices	4016 (1454)	5302	2902
Clocks/Block	12	6	Not Published
Cipher Mode	ECB or CBC	CBC	ECB or CBC
Max. Clock Frequency	50.0 MHz	14.1 MHz	Not Published

	This Design	Reference [15]	Reference [16]
Throughput	533.33 Mbps	300.1 Mbps	331.5 Mbps
Throughput/ Slice	0.133 (0.367)	0.057	0.114

Note that second version of the space-optimised design utilizing the T-BOX approach was also completed, however, this version suffered from the fact it required 48 rather than 16 ROMs. The total number of required slices increased from 4016 to 6185, a 54% increase. However, this increase in size did not translate into increased throughput. In fact, throughput decreased to 627.45 Mbps, based on a 17 ns minimum clock period. It is believed that the throughput decreased with the T-BOX approach (when one would have expected it to increase) due to the difficulty of optimising delays for 48 ROMs. The "outer-region" ROMs will have much higher net delays than those closer to the destination processing blocks. The ROM(s) with the highest delay will dominate the overall Cipher Round delay.

8.2 Multi-Session Pipelined AES Design Results

A design summary of the Multi-Session Pipelined AES design is presented in Table 10. Note that this design utilized 13675 Xilinx FPGA slices with an equivalent gate count of 262K gates.

Table 10 Multi-Session Pipelined AES Design Summary

Parameter	Complete AES Cipher
256x8-bit ROM	160
Number of Slices	13675
Equivalent Gate Count	262,073

Table 11 details the performance characteristics of the Multi-Session Pipelined AES design and compares the results with the space optimised design as well as other published results. The 10x speedup over the "space-optimised" design comes at a cost of 3.4x the total number of FPGA slices. Note that the while the aggregate throughput across all sessions is 6.4 Gbps, the throughput for any one of the concurrent sessions (in CBC mode) is 581.8 Mbps.

Note that Table 11 compares the Multi-Session Pipelined design with another design [23] that is also fully-pipelined, and on the surface offer much greater efficiency and throughput. However, it is important to note that these designs do not appear to support CBC mode, which is a mandatory mode for any network application using AES with IPSec [14]. As such, a design that fails to support CBC is of limited practical value.

Table 11 Performance Characteristics of the Multi-Session Pipelined AES Design

	This Design (Multi-Session Pipeline)	This Design (Space)	Reference [21]	Reference [23]
FPGA Type	XC2V4000-BF957-6	XCV1000 EFG860-8	XCV812E-BG560	XC2VP20-7
FPGA Slices	13675 (1165 for Rounds 1-10)	4016 (1454)	3046	9446

FPGA BRAMs	0	0	280 of 280	0
Clocks/Block	1	12	Not Published	Not Published
Cipher Mode	ECB or CBC	ECB or CBC	ECB, CBC is unknown	ECB
Max. Clock Frequency	50.0 MHz	59.70 MHz	61 MHz	Not Published
Aggregate Throughput	6.40 Gbps	636.82 Mbps	1.95 Gbps	21.64 Gbps
Aggregate Throughput/ Slice	0.468	0.159 (0.438)	0.64	2.29

Note that in [23], the authors list results for another version which utilized 84 BRAMs and 5177 slices to achieve a throughput of 21.54 Gbps.

8.3 FPGA, ASIC and Full Custom Design Results

As mentioned previously, the throughput of the Cipher is intimately tied to the logic delay of each round. Various prior works have shown that the largest component of delay is caused by the SubBytes substitution [13], and [20] – [24]. Reducing the delay increases the throughput of the design. The designs produced for this Thesis focused on ROM and look-up table implementations of SubBytes which are most amenable to FPGA-based designs. FPGAs and their synthesis tools offer a relatively simple design environment, but this comes at the cost of reduced flexibility in design approach.

ASIC and Full Custom based implementations have much greater freedom to implement non-standard cell based approaches that can optimise down to the transistor level if desired. Implementations in this area have focused on more innovative ways to reduce

the delay associated with the SubBytes [13], and [20] – [24] process, including the Binary Decision Diagram (BDD) and Twisted Binary Decision Diagram (T-BDD) discussed in [13].

Binary decision diagrams (BDD) have been shown to reduce the delay, but the methods used incur high fanin/fanout loads [13]. The "Twisted BDD" (TBDD) approach buffers and shifts the order of inputs to each output bit of the S-BOX. This approach is the fastest reported so far, but is also the highest gate count method [13].

In [19], we describe a new method known as the L-BOX that uses novel logic minimization and decoding to reduce fanin and fanout to produce a SubBytes process that minimizes Nand2 equivalents and delay at the same time. Table 12 compares the results obtained using the L-Box approach with other SubBytes optimisation approaches.

Table 12 Comparison of ASIC Speed and Size Requirements

Method	Delay (ps)	Nand2s
Finite Field [13]	2190	354-406
BDD [13]	680	2426
TBDD [13]	440	2815
L-Box, Single [19]	460	536
L-Box, Differential [19]	420	738

8.4 Summary of Results

The results of section 8.1 indicate that the space optimised AES has a 92% higher throughput, and the highest efficiency, of the cited work for both ECB and CBC mode.

The Multi-Session Pipelined design discussed in section 8.2 offers a dramatically higher throughput for both ECB and CBC modes. The Multi-Session Pipelined is capable of an aggregate throughput of 6.4 Gbps. Note that the throughput in CBC mode for any one of the concurrent sessions is 581.8 Mbps. Efficiency increased to 0.468.

Other papers [20] – [24] claim extraordinary throughputs using FPGA design approaches. Typically implemented using fully pipelined architectures, these papers appear to only support ECB mode, which is a serious shortcoming. Further, many of the comparisons that are being done are across multiple FPGA types and speed grades which lead to very misleading results.

CHAPTER 9 REALIZATION OF A SECURITY CO-PROCESSOR

The AES Cipher Module, AES Inverse Cipher, and AES Key Generation modules can be integrated together in order to realize a full AES crypto processor. Figure 38 depicts a block diagram of such a design.

The input data, output data, clock, and reset signals of all three modules share a common bus to the external world. The individual rdb, fullb, emptyb, and wrb signals are kept separate so as to allow individual monitoring and selection of the cipher and inverse cipher modules.

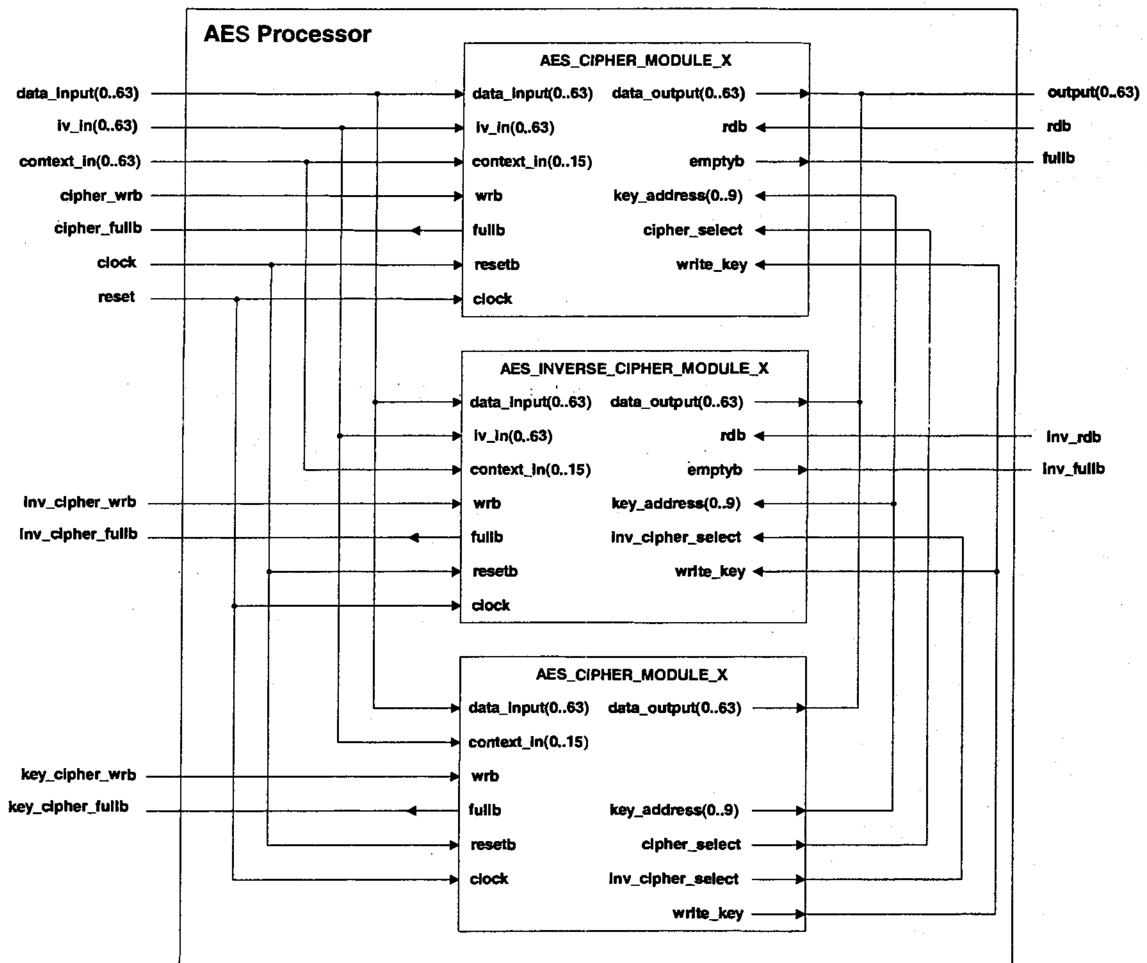
The output of the key generation module is connected to the cipher and inverse cipher modules in order to allow the round keys to be automatically updated as required. The key memory in the cipher and inverse cipher should be implemented as a dual port RAM in order to allow keys that are not in use to be updated while the cipher and inverse cipher are using other keys.

This co-processor would be capable of supporting CBC and ECB mode for both encryption and decryption, and would contain the necessary key generation logic.

A device such as the PMC-Sierra RM7000 MIPS-based processor could be used to implement the IP layer, and the IPSec protocol processing stack. Another option would be to integrate the security engine with a processor in a System on Chip (SOC) design. The small size (~79K gates) of the space-optimised design would be ideal as the die cost of the engine would be insignificant compared to the processor itself. As well, the

performance of such an integrated processor would likely be greater than an FPGA-based design. In general, ASICs offer higher performance than FPGAs (even if using the same technology, such as 0.18 uM). The VHDL code developed for this thesis is technology independent, allowing it to be synthesized in any FPGA or ASIC technology.

Figure 38 Block Diagram of the Complete AES Processor



CHAPTER 10 CONCLUSION

With more and more sensitive information being transmitted electronically over the Internet, never before has the need for strong cryptographic security been higher. In addition, as the amount and variety of devices connecting to the Internet increases, so to does the need for security processors that are tailored to the application. A security engine in a mobile phone will require vastly different performance and power specifications than a security engine operating on a core router line card.

This Thesis has explored the driving needs for security, its implementation via IPSec at the network layer, and the cryptographic protocols that form the heart of the security engine. The goal of this Thesis was to understand the issues in the design and implementation of a scalable and efficient security co-processor capable of supporting encryption and decryption at OC-12 data rates (622 Mbps). This goal has been met.

AES Cipher, Inverse Cipher (both supporting CBC and ECB mode) and Key Generation modules were completed, and verified. The code was designed in a technology independent manner, allowing it to be applied equally effectively to FPGAs or ASICs. The AES Cipher was studied to reveal some of the architectural and algorithmic optimisations that should be considered in order to address the larger speed vs. area question. In addition, a novel architecture was proposed to enable the use of pipelined architectures in CBC mode.

The space-optimised design was found to require 4016 Xilinx FPGA slices and operated at 636 Mbps, which was greater than the works cited in this Thesis. The Multi-Session

Pipelined AES design utilized a novel pipelined architecture that allowed the throughput to increase to 6.40 Gbps at the cost of an increase in FPGA slices to 13675.

There are several opportunities for future work as a result of this Thesis. The Multi-Session Pipelined approach offers multiple optimisation directions, including incorporating it coupled with a loop-unrolled architecture. As well, additional time may be spent optimising the SubBytes process, perhaps through the use of Galois field mathematics to reduce the delay instead of ROMs or LUTs. Finally, [19] describes a novel logic minimization and decoding technique which could be advanced in the full-custom arena.

APPENDIX A – SIMULATION RESULTS

Figure 39 presents a complete waveform of the space-optimised AES Cipher. The waveform was generated using the post place and route simulation model. The simulation is running the test vectors specified in Table 6. The clock is running with a period of 17 ns. As can be seen, the design produces the correct ciphertext results in 12 clock cycles per vector. This particular design includes the use of the Output FIFO, and therefore the 128 bit result is output as two 64 bit words. The first of the 64 bit outputs is only present for one clock cycle, and therefore is difficult to see. Using the rdy_counter signal as a guide, the ECB vectors are input during rdy_counter cycles 0x4 and 0x5, while the ECB ciphertext result is output during cycles 0x1A and 0x1B. Likewise, the CBC vectors are input during rdy_counter cycles 0x7-0xA, and the CBC ciphertext results are output during cycles 0x26, 0x27, 0x31 and 0x32. Figure 40, Figure 41, and Figure 42 closer views of the ciphertext results in order to verify correct operation and timing.

Figure 43 presents the simulation result for the Multi-Session Pipelined design. Using rdy_counter as a guide, the ECB test vector of Table 6 is transferred to the Cipher during cycle 0x4. The Cipher produces the result during cycle 0x10. The CBC input vectors are loaded during cycles 0x7 and 0x12. The encrypted result is presented during cycles 0x13 and 0x1E. Figure 44, Figure 45, and Figure 46, show closer views of the ciphertext results in order to verify correct operation and timing.

Figure 39 Simulation Result of the Space Optimised Cipher (Full View)

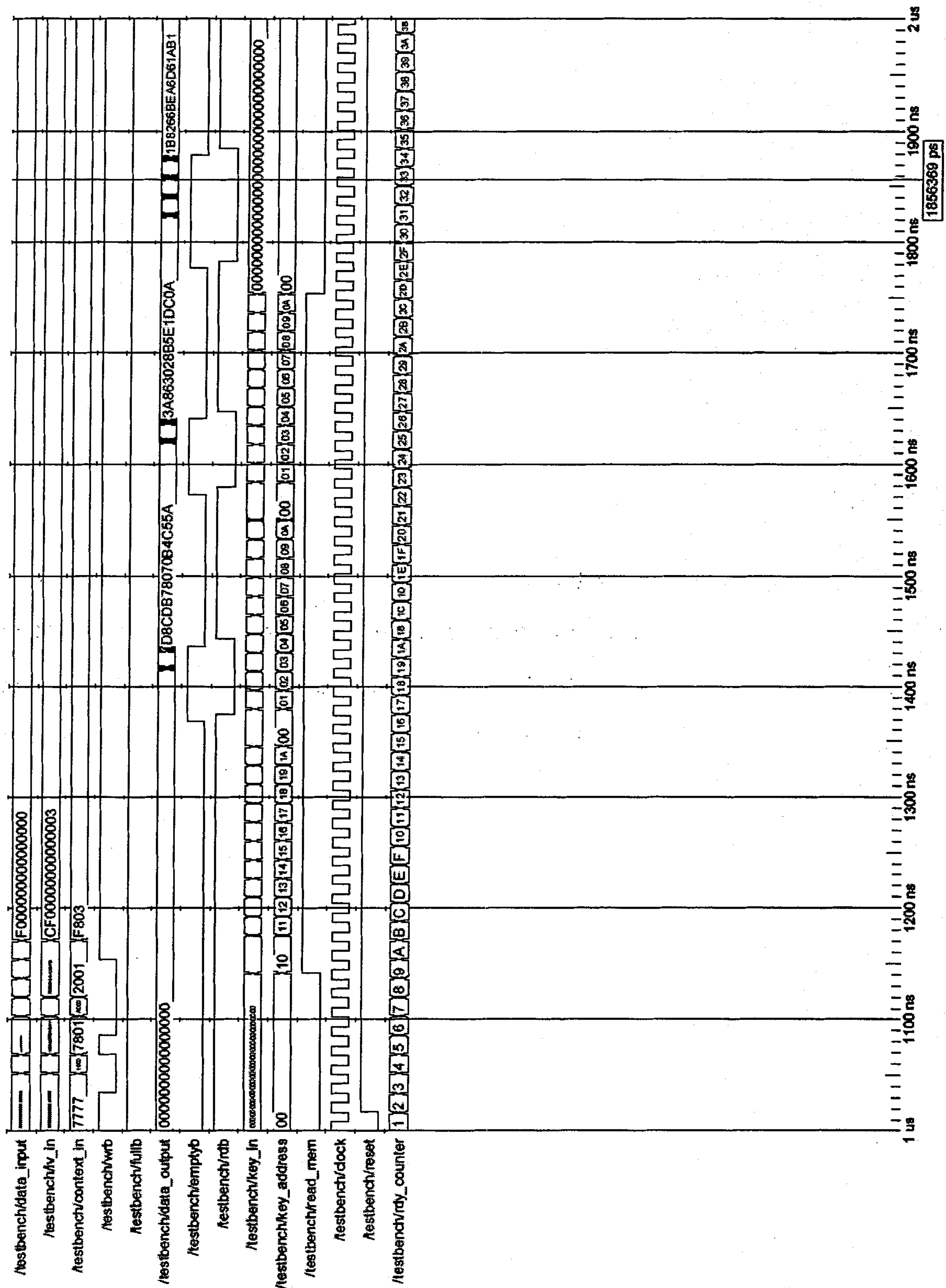


Figure 40 Simulation Result of the Space Optimised Cipher (ECB Section)

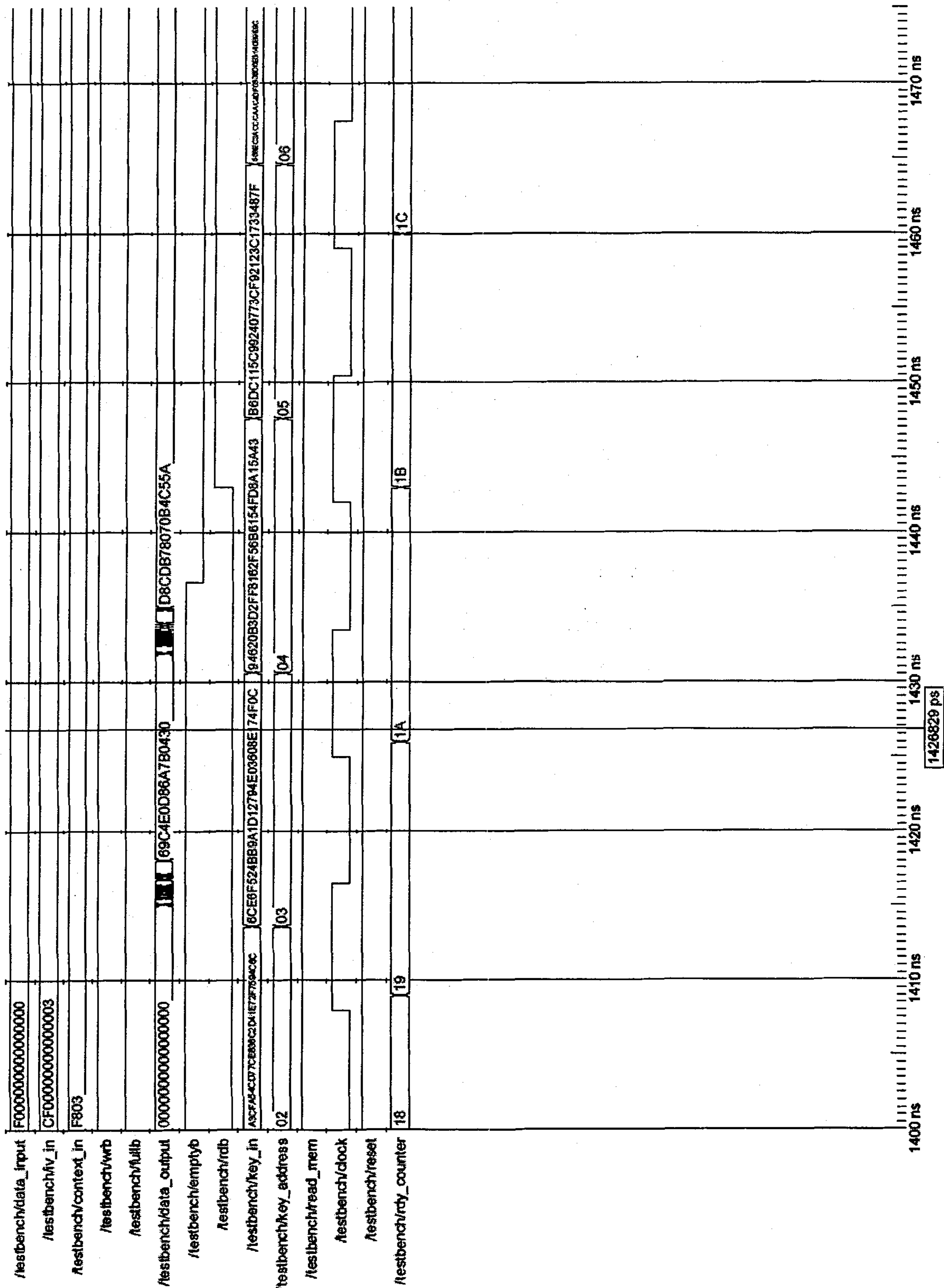


Figure 41 Simulation Result of the Space Optimised Cipher (CBC Section)

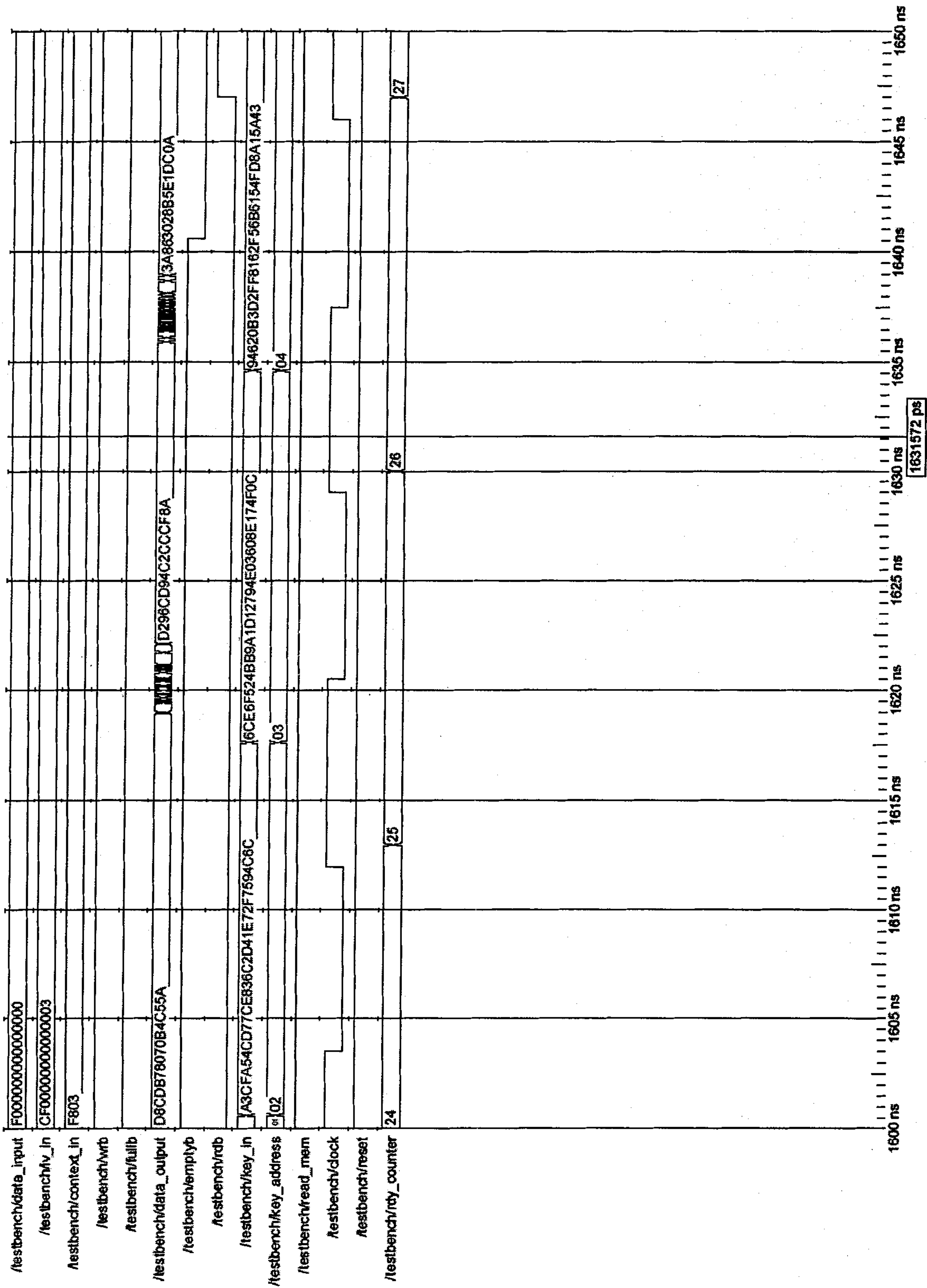


Figure 42 Simulation Result of the Space Optimised Cipher (CBC Section, Part 2)

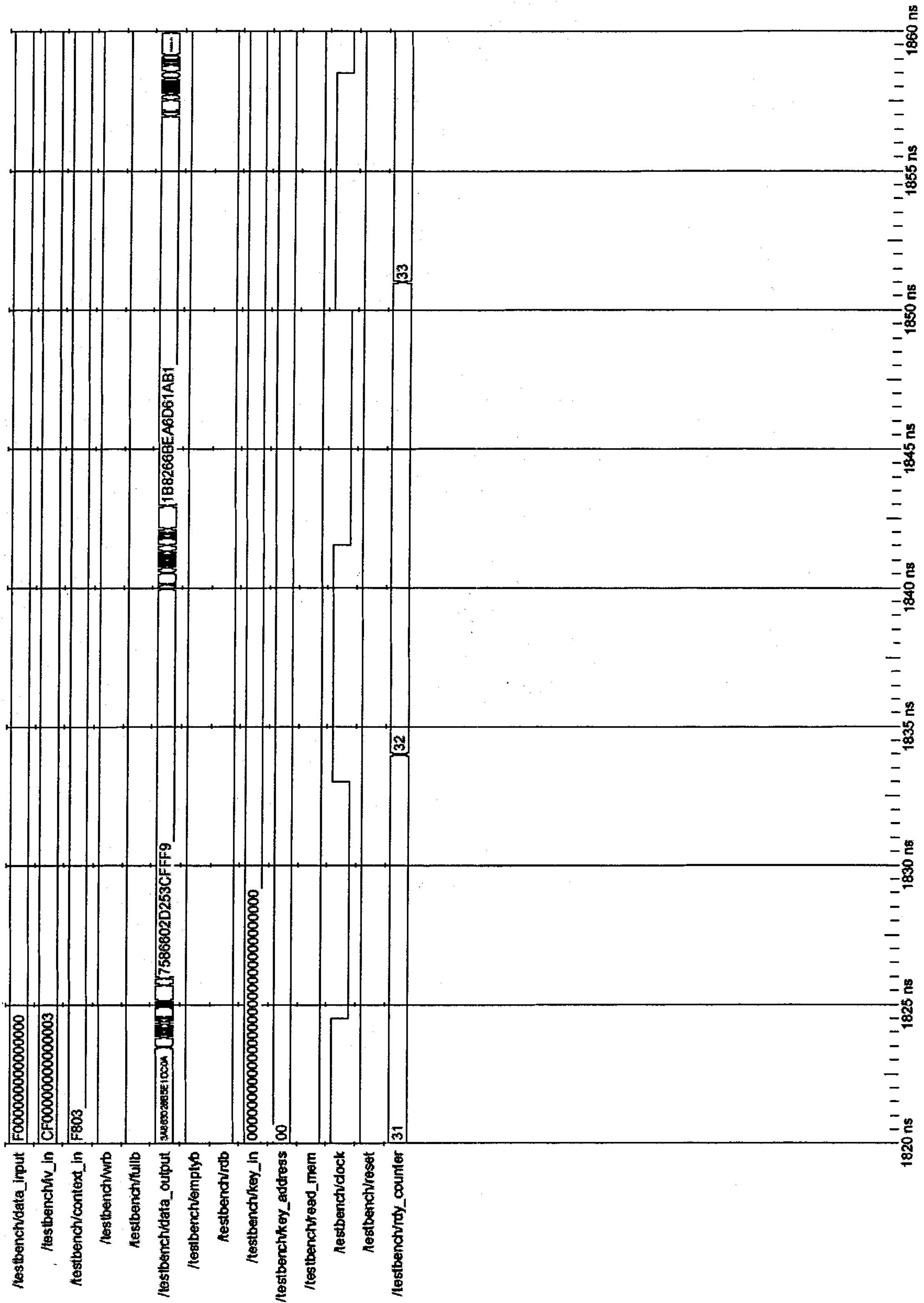


Figure 43 Simulation Result of the Multi-Session Pipelined Cipher (Full View)

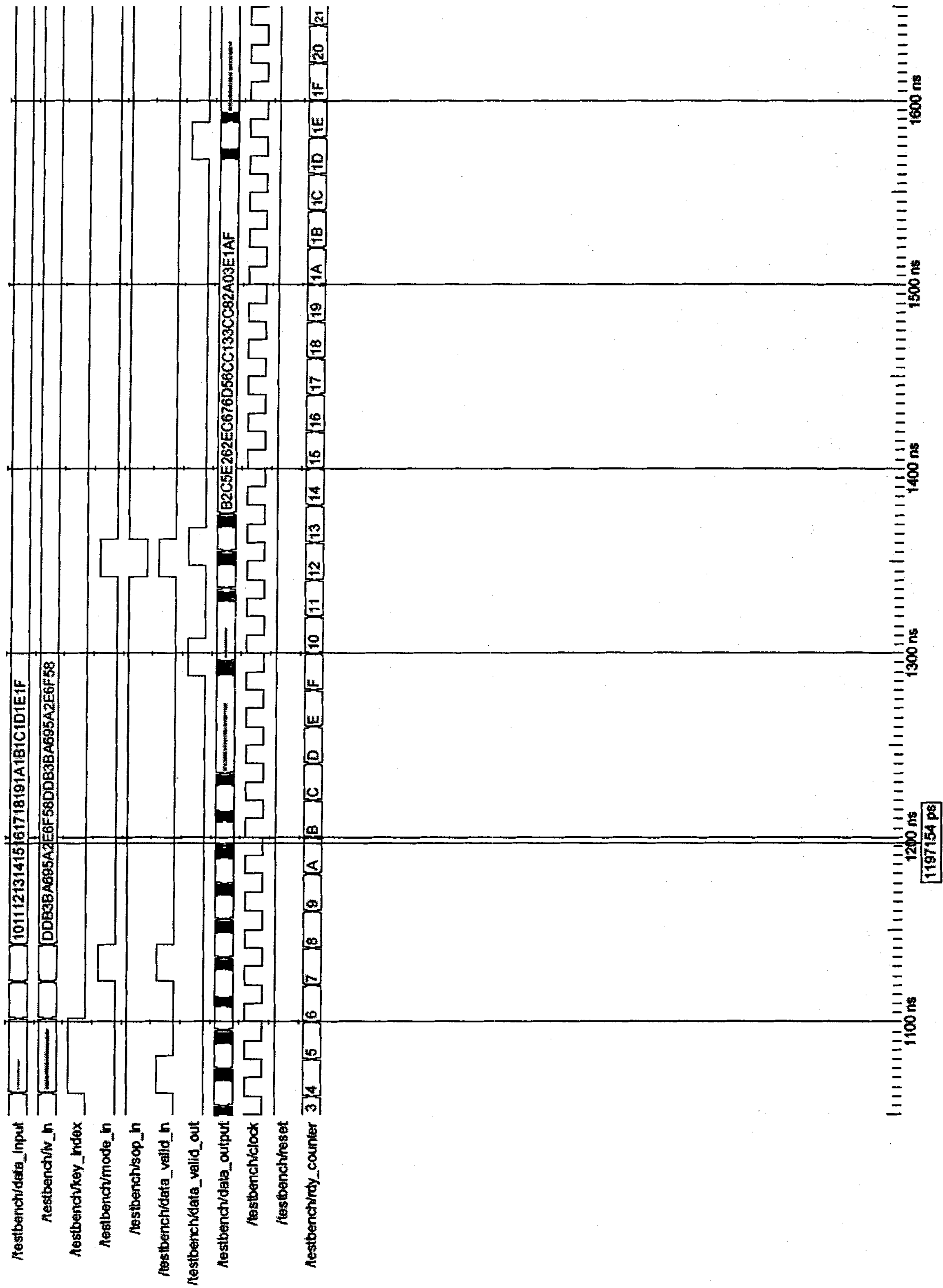


Figure 44 Simulation Result of the Multi-Session Pipelined Cipher (Inputs)

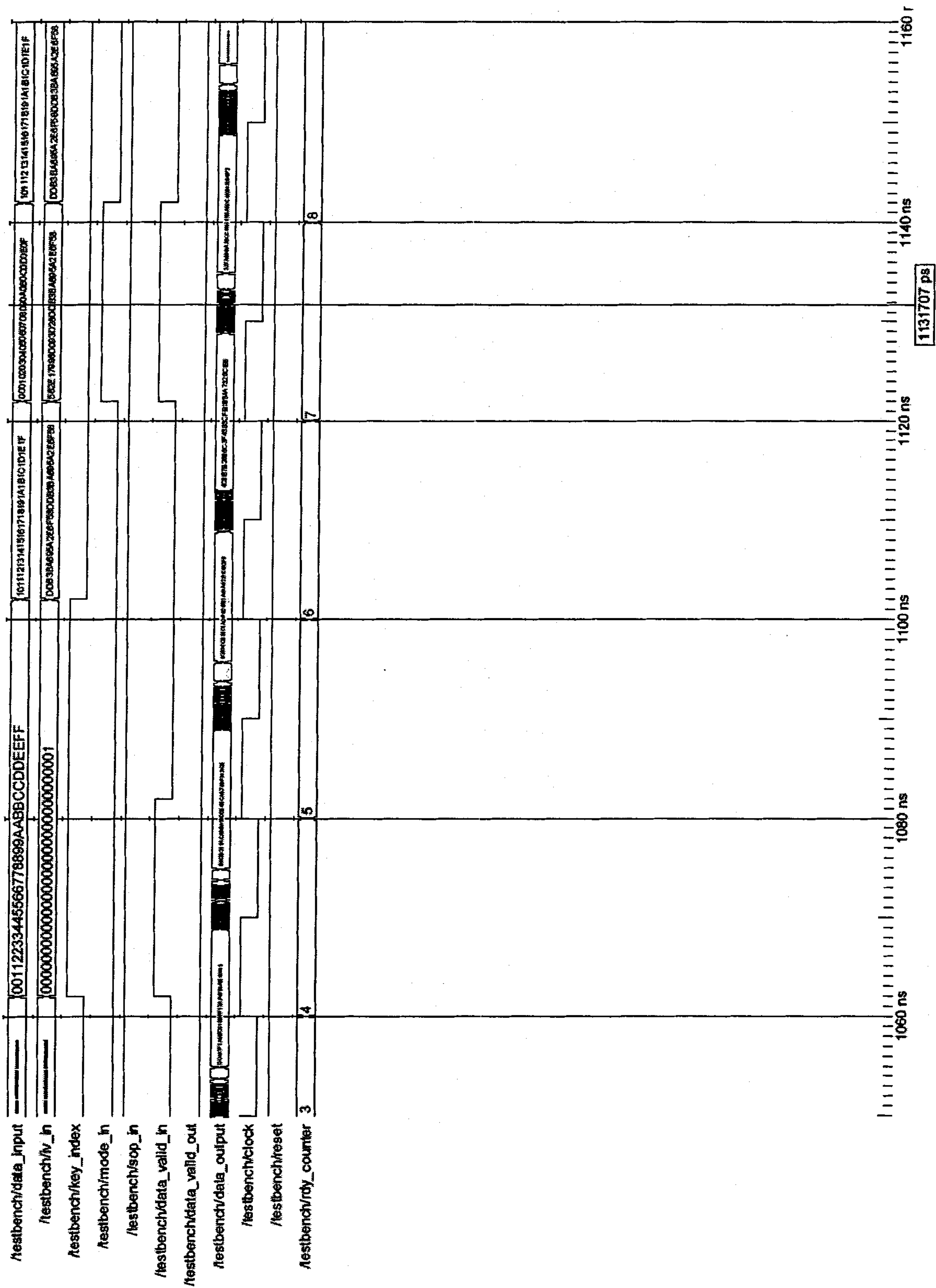


Figure 45 Simulation Result of the Multi-Session Pipelined Cipher (ECB and CBC outputs)

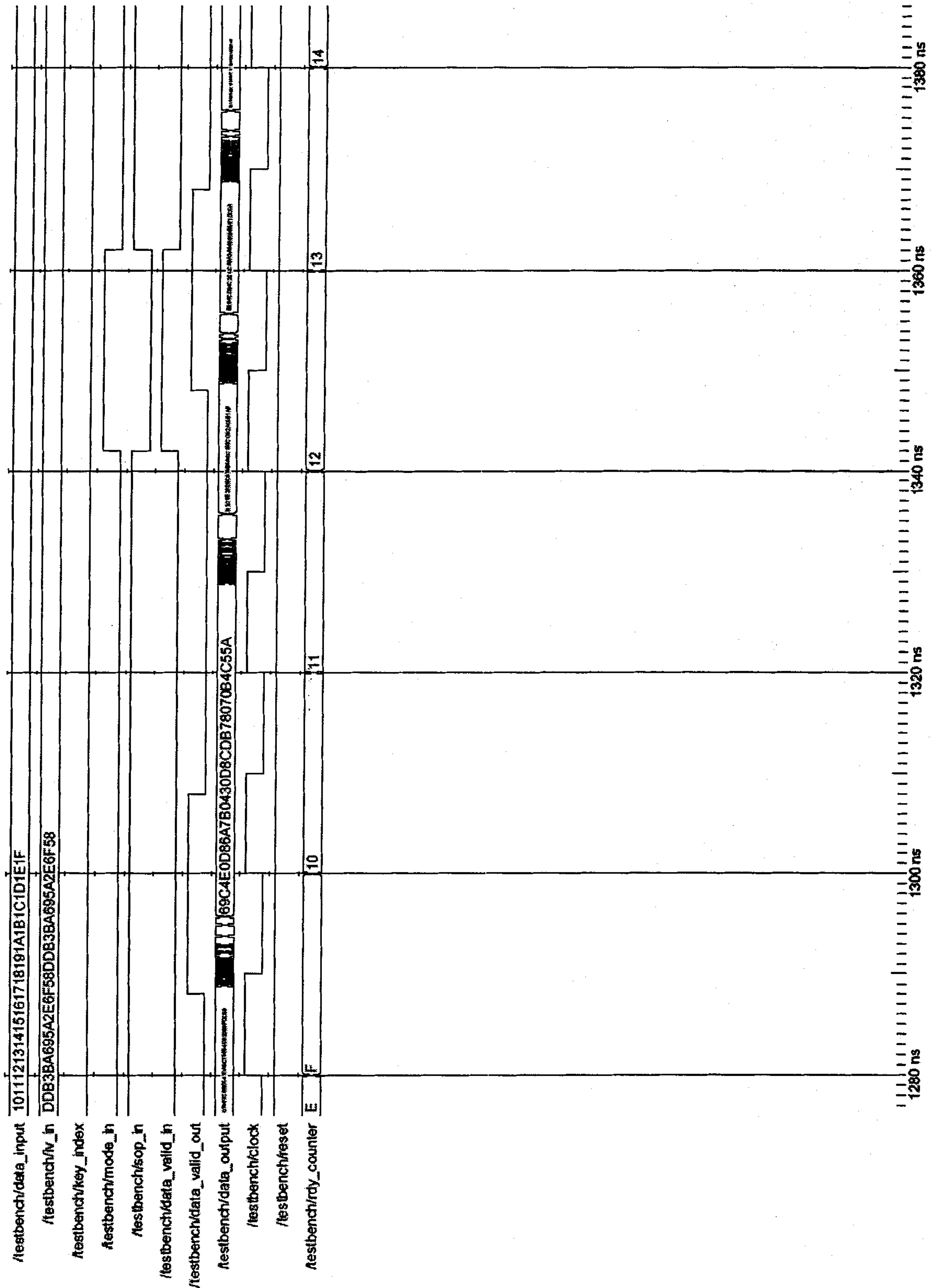
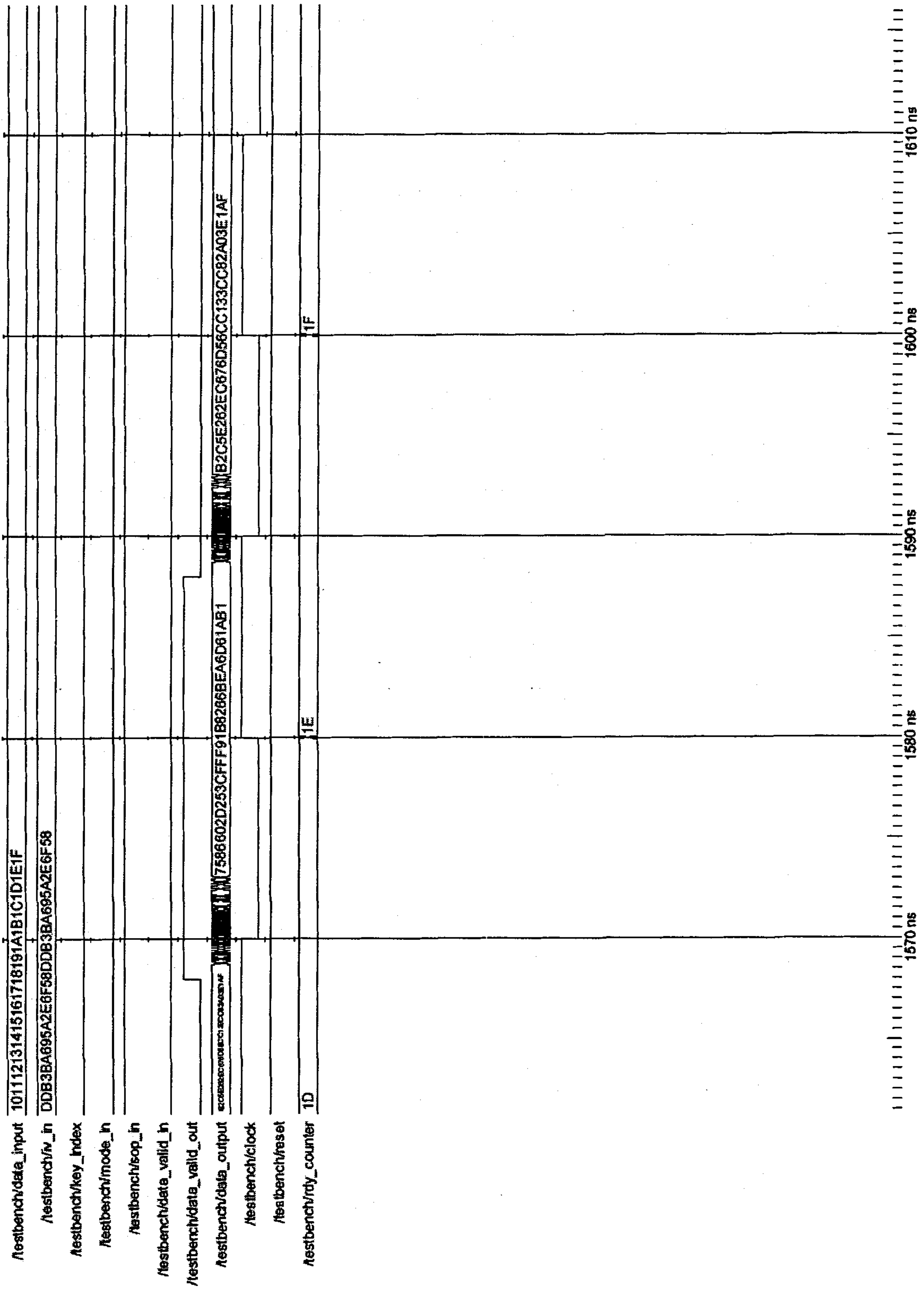


Figure 46 Simulation Result of the Multi-Session Pipelined Cipher (Last CBC output)



APPENDIX B – RTL CODE

This section presents the VHDL code of the space-optimised AES Cipher module.

AES CIPHER MODULE

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity aes_cipher_module_3 is
  Port (
                        -- i/f to input fifo
      data_input : in std_logic_vector(0 to 63);
      iv_in : in std_logic_vector(0 to 63);
      context_in : in std_logic_vector(0 to 15);
      wrb : in std_logic;
      fullb : out std_logic;

                        --i/f to output fifo
      data_output : out std_logic_vector(0 to 63);
      emptyb : out std_logic;
      rdb : in std_logic;

                        --i/f to key memory
      key_in : in std_logic_vector(0 to 127);
      key_address : out std_logic_vector(0 to 4);
      read_mem : out std_logic;

                        clock : in std_logic;
                        reset : in std_logic);
end aes_cipher_module_3;

architecture RTL of aes_cipher_module_3 is
COMPONENT control_sm
  Port (
                        -- i/f to input fifo
                        data_in : in std_logic_vector(0 to 127);
      iv : in std_logic_vector(0 to 127);
      context : in std_logic_vector(0 to 31);
      fifo_emptyb : in std_logic;
      rdb_fifo : out std_logic;
                        -- i/f to output fifo

      wrb_fifo : out std_logic;
      fifo_fullb : in std_logic;
```

```
                    -- i/f to cipher block
     aes_data_in : out std_logic_vector(0 to 127);
     round : out std_logic_vector(0 to 3);
     aes_data_out_round0 : in std_logic_vector(0 to 127);
            aes_data_out_mid : in std_logic_vector(0 to 127);
        aes_data_out_final : in std_logic_vector(0 to 127);
     aes_data_out_last : in std_logic_vector(0 to 127);

                    -- i/f to key memory
     aes_key_mem_address : out std_logic_vector(0 to 4);
     read_key_mem : out std_logic;

                    clock : in std_logic;
                    reset : in std_logic);
END COMPONENT;

COMPONENT FIFO
  Port ( resetb : in std_logic;
     clock : in std_logic;
     rdb : in std_logic;
     wrb : in std_logic;
     data_in : in std_logic_vector(0 to 63);
        iv_in : in std_logic_vector (0 to 63);
        context_in : in std_logic_vector (0 to 15);
     emptyb : out std_logic;
     fullb : out std_logic;
           context_out : out std_logic_vector (0 to 31);
           iv_out : out std_logic_vector (0 to 127);
     data_out : out std_logic_vector(0 to 127));
END COMPONENT;

COMPONENT OUT_FIFO
  Port ( resetb : in std_logic;
     clock : in std_logic;
     rdb : in std_logic;
     wrb : in std_logic;
     data_in : in std_logic_vector(0 to 127);
     emptyb : out std_logic;
     fullb : out std_logic;
     data_out : out std_logic_vector(0 to 63));
END COMPONENT;

COMPONENT aes_cipher
  Port ( aes_data_in : in std_logic_vector(0 to 127);      --data block to encrpyt
       aes_key_in : in std_logic_vector(0 to 127);     --the key to use for this round
       round_num: in std_logic_vector(0 to 3);
                    clock: in std_logic;
                    reset: in std_logic;
        aes_data_out_round0 : out std_logic_vector(0 to 127);
                    aes_data_out_mid : out std_logic_vector(0 to 127);
                    aes_data_out_final : out std_logic_vector(0 to 127);
        aes_data_out_last : out std_logic_vector(0 to 127));
END COMPONENT;

signal aes_module_sm_data_in: std_logic_vector (0 to 127);
```

```
signal aes_module_sm_context: std_logic_vector (0 to 31);
signal aes_module_sm_iv: std_logic_vector (0 to 127);
signal aes_module_sm_read: std_logic;
signal aes_module_sm_empty: std_logic;

signal aes_module_outfifo_wrb: std_logic;
signal aes_module_outfifo_fullb: std_logic;

signal aes_module_cipher_aes_data_in : std_logic_vector(0 to 127);
signal aes_module_cipher_round : std_logic_vector(0 to 3);
signal aes_module_cipher_aes_data_out_round0 : std_logic_vector(0 to 127);
signal aes_module_cipher_aes_data_out_mid : std_logic_vector(0 to 127);
signal aes_module_cipher_aes_data_out_final : std_logic_vector(0 to 127);
signal aes_module_cipher_aes_data_out_last : std_logic_vector(0 to 127);

signal aes_module_clock: std_logic;
signal aes_module_reset: std_logic;

begin

controller: control_sm PORT MAP(
                data_in => aes_module_sm_data_in,
   iv => aes_module_sm_iv,
   context => aes_module_sm_context,
   fifo_emptyb => aes_module_sm_empty,
   rdb_fifo => aes_module_sm_read,

                          -- i/f to output fifo
   wrb_fifo => aes_module_outfifo_wrb,
   fifo_fullb => aes_module_outfifo_fullb,

                          -- i/f to cipher block
   aes_data_in => aes_module_cipher_aes_data_in,
   round => aes_module_cipher_round,
   aes_data_out_round0 => aes_module_cipher_aes_data_out_round0,
        aes_data_out_mid => aes_module_cipher_aes_data_out_mid,
        aes_data_out_final => aes_module_cipher_aes_data_out_final,
        aes_data_out_last => aes_module_cipher_aes_data_out_last,

                          -- i/f to key memory
   aes_key_mem_address => key_address,
   read_key_mem => read_mem,

        clock => aes_module_clock,
        reset => aes_module_reset
       );

input_fifo: FIFO PORT MAP(
   rdb => aes_module_sm_read,
   wrb => wrb,
   data_in => data_input,
        iv_in => iv_in,
        context_in => context_in,
   emptyb => aes_module_sm_empty,
   fullb => fullb,
        context_out => aes_module_sm_context,
```

```
            iv_out => aes_module_sm_iv,
        data_out => aes_module_sm_data_in,
            clock => aes_module_clock,
            resetb => aes_module_reset
           );

    output_fifo: OUT_FIFO PORT MAP(
        rdb => rdb,
        wrb => aes_module_outfifo_wrb,
            data_in => aes_module_cipher_aes_data_out_last,
        emptyb => emptyb,
        fullb => aes_module_outfifo_fullb,
        data_out => data_output,
            clock => aes_module_clock,
            resetb => aes_module_reset
           );


    cipher: aes_cipher PORT MAP(
        aes_data_in => aes_module_cipher_aes_data_in,     --data block to encrpyt
            aes_key_in => key_in,     --the key to use for this round
            round_num => aes_module_cipher_round,
            clock => aes_module_clock,
            reset => aes_module_reset,
        aes_data_out_round0 => aes_module_cipher_aes_data_out_round0,
            aes_data_out_mid => aes_module_cipher_aes_data_out_mid,
            aes_data_out_final => aes_module_cipher_aes_data_out_final,
            aes_data_out_last => aes_module_cipher_aes_data_out_last
           );

    aes_module_clock <= clock;
    aes_module_reset <= reset;

end RTL;
```

AES CIPHER – S-BOX Approach

```
    library IEEE;
    use IEEE.STD_LOGIC_1164.ALL;
    use IEEE.STD_LOGIC_ARITH.ALL;
    use IEEE.STD_LOGIC_UNSIGNED.ALL;

    entity aes_cipher is
      Port ( aes_data_in : in std_logic_vector(0 to 127);        --data block to encrpyt
                        aes_key_in : in std_logic_vector(0 to 127);      --the key to use
for this round
                        round_num: in std_logic_vector(0 to 3);
                        clock: in std_logic;
                        reset: in std_logic;
                        aes_data_out_round0 : out std_logic_vector(0 to 127);
                        aes_data_out_mid : out std_logic_vector(0 to 127);
                        aes_data_out_final : out std_logic_vector(0 to 127);
            aes_data_out_last : out std_logic_vector(0 to 127));       --output block of data
from this round
    end aes_cipher;
```

```
architecture rtl of aes_cipher is

signal aes_key_in_R: std_logic_vector (0 to 127);

signal round_num_R: std_logic_vector (0 to 3);
signal aes_data: std_logic_vector (0 to 127);

--signal output_valid_I: std_logic;

type state is array (0 to 15) of std_logic_vector(0 to 7);
signal instate: state;
signal substate: state;
signal shiftstate: state;
signal shiftstate_2: state;
signal outmixState: state;

signal shift_data_out: std_logic_vector(0 to 127);
signal mix_data_out: std_logic_vector(0 to 127);

signal last_round_out: std_logic_vector(0 to 127);

--following is output of MixColumns() (in state format (dbyteROWCOLUMN)

signal OutMixByte00: std_logic_vector(0 to 7);
signal OutMixByte01: std_logic_vector(0 to 7);
signal OutMixByte02: std_logic_vector(0 to 7);
signal OutMixByte03: std_logic_vector(0 to 7);

signal OutMixByte10: std_logic_vector(0 to 7);
signal OutMixByte11: std_logic_vector(0 to 7);
signal OutMixByte12: std_logic_vector(0 to 7);
signal OutMixByte13: std_logic_vector(0 to 7);

signal OutMixByte20: std_logic_vector(0 to 7);
signal OutMixByte21: std_logic_vector(0 to 7);
signal OutMixByte22: std_logic_vector(0 to 7);
signal OutMixByte23: std_logic_vector(0 to 7);

signal OutMixByte30: std_logic_vector(0 to 7);
signal OutMixByte31: std_logic_vector(0 to 7);
signal OutMixByte32: std_logic_vector(0 to 7);
signal OutMixByte33: std_logic_vector(0 to 7);

subtype S_BOX_FIELD      is integer range 0 to 255;
subtype SBOX_INDEX_TYPE  is integer range 0 to 15;
type SBOX_TYPE      is array (0 to 255) of S_BOX_FIELD;
constant SBOXs : SBOX_TYPE := (

 99, 124, 119, 123, 242, 107, 111, 197,  48,   1, 103,  43, 254, 215, 171, 118,
202, 130, 201, 125, 250,  89,  71, 240, 173, 212, 162, 175, 156, 164, 114, 192,
183, 253, 147,  38,  54,  63, 247, 204,  52, 165, 229, 241, 113, 216,  49,  21,
  4, 199,  35, 195,  24, 150,   5, 154,   7,  18, 128, 226, 235,  39, 178, 117,
  9, 131,  44,  26,  27, 110,  90, 160,  82,  59, 214, 179,  41, 227,  47, 132,
 83, 209,   0, 237,  32, 252, 177,  91, 106, 203, 190,  57,  74,  76,  88, 207,
208, 239, 170, 251,  67,  77,  51, 133,  69, 249,   2, 127,  80,  60, 159, 168,
 81, 163,  64, 143, 146, 157,  56, 245, 188, 182, 218,  33,  16, 255, 243, 210,
```

```
205, 12, 19, 236, 95, 151, 68, 23, 196, 167, 126, 61, 100, 93, 25, 115,
 96, 129, 79, 220, 34, 42, 144, 136, 70, 238, 184, 20, 222, 94, 11, 219,
224, 50, 58, 10, 73, 6, 36, 92, 194, 211, 172, 98, 145, 149, 228, 121,
231, 200, 55, 109, 141, 213, 78, 169, 108, 86, 244, 234, 101, 122, 174, 8,
186, 120, 37, 46, 28, 166, 180, 198, 232, 221, 116, 31, 75, 189, 139, 138,
112, 62, 181, 102, 72, 3, 246, 14, 97, 53, 87, 185, 134, 193, 29, 158,
225, 248, 152, 17, 105, 217, 142, 148, 155, 30, 135, 233, 206, 85, 40, 223,
140, 161, 137, 13, 191, 230, 66, 104, 65, 153, 45, 15, 176, 84, 187, 22

);

function ESubBytes (inbyte: std_logic_vector(0 to 7)) return std_logic_vector is
variable return_val: std_logic_vector(0 to 7);
begin

return conv_std_logic_vector(SBOXs(conv_integer(inbyte)), 8);
end function;

function DubBytes (inbyte: std_logic_vector(0 to 7)) return std_logic_vector is
begin

case inbyte(0) is
    when '0' =>
      return inbyte(1 to 7) & '0';
    when '1' =>
      return ((inbyte(1 to 7) & '0') xor "00011011");
    when others =>
      return "00000000";
  end case;
end function;

begin

aes_data <= aes_data_in;
aes_key_in_R <= aes_key_in;
round_num_R <= round_num;

--place the input data in the state, as defined in the AES spec.
instate(0) <= aes_data(0 to 7);
instate(1) <= aes_data(8 to 15);
instate(2) <= aes_data(16 to 23);
instate(3) <= aes_data(24 to 31);

instate(4) <= aes_data(32 to 39);
instate(5) <= aes_data(40 to 47);
instate(6) <= aes_data(48 to 55);
instate(7) <= aes_data(56 to 63);

instate(8) <= aes_data(64 to 71);
instate(9) <= aes_data(72 to 79);
instate(10) <= aes_data(80 to 87);
instate(11) <= aes_data(88 to 95);

instate(12) <= aes_data(96 to 103);
instate(13) <= aes_data(104 to 111);
instate(14) <= aes_data(112 to 119);
```

```
instate(15) <= aes_data(120 to 127);

--perform the SubBytes function on all bytes of the state.

substate(0) <= ESubBytes(instate(0));
substate(1) <= ESubBytes(instate(1));
substate(2) <= ESubBytes(instate(2));
substate(3) <= ESubBytes(instate(3));

substate(4) <= ESubBytes(instate(4));
substate(5) <= ESubBytes(instate(5));
substate(6) <= ESubBytes(instate(6));
substate(7) <= ESubBytes(instate(7));

substate(8) <= ESubBytes(instate(8));
substate(9) <= ESubBytes(instate(9));
substate(10) <= ESubBytes(instate(10));
substate(11) <= ESubBytes(instate(11));

substate(12) <= ESubBytes(instate(12));
substate(13) <= ESubBytes(instate(13));
substate(14) <= ESubBytes(instate(14));
substate(15) <= ESubBytes(instate(15));

--perform the ShiftRows function on all rows of the state.

shiftstate(0) <= substate(0);
shiftstate(4) <= substate(4);
shiftstate(8) <= substate(8);
shiftstate(12) <= substate(12);

shiftstate(1) <= substate(5);
shiftstate(5) <= substate(9);
shiftstate(9) <= substate(13);
shiftstate(13) <= substate(1);

shiftstate(2) <= substate(10);
shiftstate(6) <= substate(14);
shiftstate(10) <= substate(2);
shiftstate(14) <= substate(6);

shiftstate(3) <= substate(15);
shiftstate(7) <= substate(3);
shiftstate(11) <= substate(7);
shiftstate(15) <= substate(11);

--for the last round, the output is the xor of the state after shiftrows, and the key,
--so create the last round output word

shift_data_out(0 to 7) <= shiftstate(0);
shift_data_out(8 to 15) <= shiftstate(1);
shift_data_out(16 to 23) <= shiftstate(2);
shift_data_out(24 to 31) <= shiftstate(3);

shift_data_out(32 to 39) <= shiftstate(4);
shift_data_out(40 to 47) <= shiftstate(5);
```

```
shift_data_out(48 to 55) <= shiftstate(6);
shift_data_out(56 to 63) <= shiftstate(7);

shift_data_out(64 to 71) <= shiftstate(8);
shift_data_out(72 to 79) <= shiftstate(9);
shift_data_out(80 to 87) <= shiftstate(10);
shift_data_out(88 to 95) <= shiftstate(11);

shift_data_out(96 to 103) <= shiftstate(12);
shift_data_out(104 to 111) <= shiftstate(13);
shift_data_out(112 to 119) <= shiftstate(14);
shift_data_out(120 to 127) <= shiftstate(15);

--for mixcolumns, we need to take 1, 2, and 3 times various bytes in the columns
--the following creates the x2. x3 is the xor of x2 and the original (x1) value.

shiftstate_2(0) <= DubBytes(substate(0));
shiftstate_2(4) <= DubBytes(substate(4));
shiftstate_2(8) <= DubBytes(substate(8));
shiftstate_2(12) <= DubBytes(substate(12));

shiftstate_2(1) <= DubBytes(substate(5));
shiftstate_2(5) <= DubBytes(substate(9));
shiftstate_2(9) <= DubBytes(substate(13));
shiftstate_2(13) <= DubBytes(substate(1));

shiftstate_2(2) <= DubBytes(substate(10));
shiftstate_2(6) <= DubBytes(substate(14));
shiftstate_2(10) <= DubBytes(substate(2));
shiftstate_2(14) <= DubBytes(substate(6));

shiftstate_2(3) <= DubBytes(substate(15));
shiftstate_2(7) <= DubBytes(substate(3));
shiftstate_2(11) <= DubBytes(substate(7));
shiftstate_2(15) <= DubBytes(substate(11));

--Following groups perform the MixColumns operation, as defined in the AES standard
--OutMixByte00, OutMixByte10, OutMixByte20, OutMixByte30
OutMixByte00 <= (shiftstate(2) xor shiftstate(3) xor shiftstate_2(0) xor (shiftstate_2(1) xor
shiftstate(1)));
OutMixByte10 <= (shiftstate(0) xor shiftstate(3) xor shiftstate_2(1) xor (shiftstate_2(2) xor
shiftstate(2)));
OutMixByte20 <= (shiftstate(0) xor shiftstate(1) xor shiftstate_2(2) xor (shiftstate_2(3) xor
shiftstate(3)));
OutMixByte30 <= (shiftstate(1) xor shiftstate(2) xor shiftstate_2(3) xor (shiftstate_2(0) xor
shiftstate(0)));

--OutMixByte01, OutMixByte11, OutMixByte21, OutMixByte31
OutMixByte01 <= (shiftstate(6) xor shiftstate(7) xor shiftstate_2(4) xor (shiftstate_2(5) xor
shiftstate(5)));
OutMixByte11 <= (shiftstate(4) xor shiftstate(7) xor shiftstate_2(5) xor (shiftstate_2(6) xor
shiftstate(6)));
OutMixByte21 <= (shiftstate(4) xor shiftstate(5) xor shiftstate_2(6) xor (shiftstate_2(7) xor
shiftstate(7)));
OutMixByte31 <= (shiftstate(5) xor shiftstate(6) xor shiftstate_2(7) xor (shiftstate_2(4) xor
shiftstate(4)));
```

```
        --OutMixByte02, OutMixByte12, OutMixByte22, OutMixByte32
        OutMixByte02 <= (shiftstate(10) xor shiftstate(11) xor shiftstate_2(8) xor (shiftstate_2(9)
xor shiftstate(9)));
        OutMixByte12 <= (shiftstate(8) xor shiftstate(11) xor shiftstate_2(9) xor (shiftstate_2(10)
xor shiftstate(10)));
        OutMixByte22 <= (shiftstate(8) xor shiftstate(9) xor shiftstate_2(10) xor (shiftstate_2(11)
xor shiftstate(11)));
        OutMixByte32 <= (shiftstate(9) xor shiftstate(10) xor shiftstate_2(11) xor (shiftstate_2(8)
xor shiftstate(8)));

        --OutMixByte03, OutMixByte13, OutMixByte23, OutMixByte33
        OutMixByte03 <= (shiftstate(14) xor shiftstate(15) xor shiftstate_2(12) xor
(shiftstate_2(13) xor shiftstate(13)));
        OutMixByte13 <= (shiftstate(12) xor shiftstate(15) xor shiftstate_2(13) xor
(shiftstate_2(14) xor shiftstate(14)));
        OutMixByte23 <= (shiftstate(12) xor shiftstate(13) xor shiftstate_2(14) xor
(shiftstate_2(15) xor shiftstate(15)));
        OutMixByte33 <= (shiftstate(13) xor shiftstate(14) xor shiftstate_2(15) xor
(shiftstate_2(12) xor shiftstate(12)));

        --ollowing is the output for rounds 1-9 of the cipher
        mix_data_out(0 to 7) <= OutMixByte00;
        mix_data_out(8 to 15) <= OutMixByte10;
        mix_data_out(16 to 23) <= OutMixByte20;
        mix_data_out(24 to 31) <= OutMixByte30;

        mix_data_out(32 to 39) <= OutMixByte01;
        mix_data_out(40 to 47) <= OutMixByte11;
        mix_data_out(48 to 55) <= OutMixByte21;
        mix_data_out(56 to 63) <= OutMixByte31;

        mix_data_out(64 to 71) <= OutMixByte02;
        mix_data_out(72 to 79) <= OutMixByte12;
        mix_data_out(80 to 87) <= OutMixByte22;
        mix_data_out(88 to 95) <= OutMixByte32;

        mix_data_out(96 to 103) <= OutMixByte03;
        mix_data_out(104 to 111) <= OutMixByte13;
        mix_data_out(112 to 119) <= OutMixByte23;
        mix_data_out(120 to 127) <= OutMixByte33;

        process (clock)
        begin
         if (clock'event and clock = '1') then
                        if (round_num_R = "1011") then
                                aes_data_out_last <= shift_data_out xor aes_key_in_R;
                        end if;
         end if;
        end process;

        aes_data_out_round0 <= aes_data xor aes_key_in_R;
        aes_data_out_mid <= mix_data_out xor aes_key_in_R;
        aes_data_out_final <= shift_data_out xor aes_key_in_R;

end rtl;
```

AES CIPHER – T-BOX Approach

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity aes_cipher is
   Port ( aes_data_in : in std_logic_vector(0 to 127);      --data block to encrpyt
         aes_key_in : in std_logic_vector(0 to 127);     --the key to use for this round
         round_num: in std_logic_vector(0 to 3);
         clock: in std_logic;
         reset: in std_logic;
        aes_data_out_round0 : out std_logic_vector(0 to 127);
        aes_data_out_mid : out std_logic_vector(0 to 127);
        aes_data_out_final : out std_logic_vector(0 to 127);
        aes_data_out_last : out std_logic_vector(0 to 127));
end aes_cipher;

architecture rtl of aes_cipher is

--Registers for inputs
signal aes_key_in_R: std_logic_vector (0 to 127);

signal round_num_R: std_logic_vector (0 to 3);
signal aes_data: std_logic_vector (0 to 127);

type state is array (0 to 15) of std_logic_vector(0 to 7);
signal instate: state;

signal mid_round_out: std_logic_vector(0 to 127);
signal final_round_out: std_logic_vector(0 to 127);

signal pre1_mid_round_out: std_logic_vector(0 to 127);
signal pre2_mid_round_out: std_logic_vector(0 to 127);

signal aes_data1 : std_logic_vector(0 to 15);
signal aes_data2 : std_logic_vector(0 to 15);
signal aes_data3 : std_logic_vector(0 to 15);
signal aes_data4 : std_logic_vector(0 to 15);
signal aes_data5 : std_logic_vector(0 to 15);
signal aes_data6 : std_logic_vector(0 to 15);
signal aes_data7 : std_logic_vector(0 to 15);
signal aes_data8 : std_logic_vector(0 to 15);

subtype S_BOX_FIELD      is integer range 0 to 255;
subtype SBOX_INDEX_TYPE  is integer range 0 to 15;
type SBOX_TYPE       is array (0 to 255) of S_BOX_FIELD;
constant SBOX : SBOX_TYPE := (

 99, 124, 119, 123, 242, 107, 111, 197,  48,   1, 103,  43, 254, 215, 171, 118,
202, 130, 201, 125, 250,  89,  71, 240, 173, 212, 162, 175, 156, 164, 114, 192,
183, 253, 147,  38,  54,  63, 247, 204,  52, 165, 229, 241, 113, 216,  49,  21,
  4, 199,  35, 195,  24, 150,   5, 154,   7,  18, 128, 226, 235,  39, 178, 117,
```

```
9, 131, 44, 26, 27, 110, 90, 160, 82, 59, 214, 179, 41, 227, 47, 132,
83, 209, 0, 237, 32, 252, 177, 91, 106, 203, 190, 57, 74, 76, 88, 207,
208, 239, 170, 251, 67, 77, 51, 133, 69, 249, 2, 127, 80, 60, 159, 168,
81, 163, 64, 143, 146, 157, 56, 245, 188, 182, 218, 33, 16, 255, 243, 210,
205, 12, 19, 236, 95, 151, 68, 23, 196, 167, 126, 61, 100, 93, 25, 115,
96, 129, 79, 220, 34, 42, 144, 136, 70, 238, 184, 20, 222, 94, 11, 219,
224, 50, 58, 10, 73, 6, 36, 92, 194, 211, 172, 98, 145, 149, 228, 121,
231, 200, 55, 109, 141, 213, 78, 169, 108, 86, 244, 234, 101, 122, 174, 8,
186, 120, 37, 46, 28, 166, 180, 198, 232, 221, 116, 31, 75, 189, 139, 138,
112, 62, 181, 102, 72, 3, 246, 14, 97, 53, 87, 185, 134, 193, 29, 158,
225, 248, 152, 17, 105, 217, 142, 148, 155, 30, 135, 233, 206, 85, 40, 223,
140, 161, 137, 13, 191, 230, 66, 104, 65, 153, 45, 15, 176, 84, 187, 22

);

constant SBOX_2 : SBOX_TYPE := (
198, 248, 238, 246, 255, 214, 222, 145, 96, 2, 206, 86, 231, 181, 77, 236, 143,
31, 137, 250, 239, 178, 142, 251, 65, 179, 95, 69, 35, 83, 228, 155, 117, 225,
61, 76, 108, 126, 245, 131, 104, 81, 209, 249, 226, 171, 98, 42, 8, 149, 70,
157, 48, 55, 10, 47, 14, 36, 27, 223, 205, 78, 127, 234, 18, 29, 88, 52, 54,
220, 180, 91, 164, 118, 183, 125, 82, 221, 94, 19, 166, 185, 0, 193, 64, 227, 121,
182, 212, 141, 103, 114, 148, 152, 176, 133, 187, 197, 79, 237, 134, 154, 102, 17,
138, 233, 4, 254, 160, 120, 37, 75, 162, 93, 128, 5, 63, 33, 112, 241, 99, 119, 175,
66, 32, 229, 253, 191, 129, 24, 38, 195, 190, 53, 136, 46, 147, 85, 252, 122, 200, 186,
50, 230, 192, 25, 158, 163, 68, 84, 59, 11, 140, 199, 107, 40, 167, 188, 22, 173, 219,
100, 116, 20, 146, 12, 72, 184, 159, 189, 67, 196, 57, 49, 211, 242, 213, 139, 110, 218,
1, 177, 156, 73, 216, 172, 243, 207, 202, 244, 71, 16, 111, 240, 74, 92, 56, 87, 115, 151,
203, 161, 232, 62, 150, 97, 13, 15, 224, 124, 113, 204, 144, 6, 247, 28, 194, 106, 174,
105, 23, 153, 58, 39, 217, 235, 43, 34, 210, 169, 7, 51, 45, 60, 21, 201, 135, 170, 80,
165, 3, 89, 9, 26, 101, 215, 132, 208, 130, 41, 90, 30, 123, 168, 109, 44);

constant SBOX_3 : SBOX_TYPE := (
165, 132, 153, 141, 13, 189, 177, 84, 80, 3, 169, 125, 25, 98, 230, 154, 69, 157, 64,
135, 21, 235, 201, 11, 236, 103, 253, 234, 191, 247, 150, 91, 194, 28, 174, 106, 90,
65, 2, 79, 92, 244, 52, 8, 147, 115, 83, 63, 12, 82, 101, 94, 40, 161, 15, 181, 9, 54,
155, 61, 38, 105, 205, 159, 27, 158, 116, 46, 45, 178, 238, 251, 246, 77, 97, 206, 123,
62, 113, 151, 245, 104, 0, 44, 96, 31, 200, 237, 190, 70, 217, 75, 222, 212, 232, 74,
107, 42, 229, 22, 197, 215, 85, 148, 207, 16, 6, 129, 240, 68, 186, 227, 243, 254, 192,
138, 173, 188, 72, 4, 223, 193, 117, 99, 48, 26, 14, 109, 76, 20, 53, 47, 225, 162, 204,
57, 87, 242, 130, 71, 172, 231, 43, 149, 160, 152, 209, 127, 102, 126, 171, 131, 202, 41,
211, 60, 121, 226, 29, 118, 59, 86, 78, 30, 219, 10, 108, 228, 93, 110, 239, 166, 168,
164, 55, 139, 50, 67, 89, 183, 140, 100, 210, 224, 180, 250, 7, 37, 175, 142, 233, 24,
213,
136, 111, 114, 36, 241, 199, 81, 35, 124, 156, 33, 221, 220, 134, 133, 144, 66, 196, 170,
216, 5, 1, 18, 163, 95, 249, 208, 145, 88, 39, 185, 56, 19, 179, 51, 187, 112, 137, 167,
182, 34, 146, 32, 73, 255, 120, 122, 143, 248, 128, 23, 218, 49, 198, 184, 195, 176, 119,
17, 203, 252, 214, 58);

function SBOX2SubBytes (inbyte: std_logic_vector(0 to 7)) return std_logic_vector is
variable return_val: std_logic_vector(0 to 7);
begin

return conv_std_logic_vector(SBOX_2(conv_integer(inbyte)), 8);
end function;

function SBOX3SubBytes (inbyte: std_logic_vector(0 to 7)) return std_logic_vector is
```

```
variable return_val: std_logic_vector(0 to 7);
begin

return conv_std_logic_vector(SBOX_3(conv_integer(inbyte)), 8);
end function;

function SBOXSubBytes (inbyte: std_logic_vector(0 to 7)) return std_logic_vector is
variable return_val: std_logic_vector(0 to 7);
begin

return conv_std_logic_vector(SBOX(conv_integer(inbyte)), 8);
end function;

begin

aes_data1 <= aes_data_in(0 to 15);
aes_data2 <= aes_data_in(16 to 31);
aes_data3 <= aes_data_in(32 to 47);
aes_data4 <= aes_data_in(48 to 63);
aes_data5 <= aes_data_in(64 to 79);
aes_data6 <= aes_data_in(80 to 95);
aes_data7 <= aes_data_in(96 to 111);
aes_data8 <= aes_data_in(112 to 127);

aes_key_in_R <= aes_key_in;
round_num_R <= round_num;

--place the input data in the state, as defined in the AES spec.
instate(0) <= aes_data1(0 to 7);
instate(1) <= aes_data1(8 to 15);
instate(2) <= aes_data2(0 to 7);
instate(3) <= aes_data2(8 to 15);

instate(4) <= aes_data3(0 to 7);
instate(5) <= aes_data3(8 to 15);
instate(6) <= aes_data4(0 to 7);
instate(7) <= aes_data4(8 to 15);

instate(8) <= aes_data5(0 to 7);
instate(9) <= aes_data5(8 to 15);
instate(10) <= aes_data6(0 to 7);
instate(11) <= aes_data6(8 to 15);

instate(12) <= aes_data7(0 to 7);
instate(13) <= aes_data7(8 to 15);
instate(14) <= aes_data8(0 to 7);
instate(15) <= aes_data8(8 to 15);

        pre1_mid_round_out(0 to 7) <= SBOX2SubBytes(instate(0)) xor
SBOX3SubBytes(instate(5));
        pre1_mid_round_out(8 to 15) <= SBOXSubBytes(instate(0)) xor
SBOX2SubBytes(instate(5));
        pre1_mid_round_out(16 to 23) <= SBOXSubBytes(instate(0)) xor
SBOXSubBytes(instate(5));
        pre1_mid_round_out(24 to 31) <= SBOX3SubBytes(instate(0)) xor
SBOXSubBytes(instate(5));
```

```
        pre1_mid_round_out(32 to 39) <= SBOX2SubBytes(instate(4)) xor
SBOX3SubBytes(instate(9));
        pre1_mid_round_out(40 to 47) <= SBOXSubBytes(instate(4)) xor
SBOX2SubBytes(instate(9));
        pre1_mid_round_out(48 to 55) <= SBOXSubBytes(instate(4)) xor
SBOXSubBytes(instate(9));
        pre1_mid_round_out(56 to 63) <= SBOX3SubBytes(instate(4)) xor
SBOXSubBytes(instate(9));


        pre1_mid_round_out(64 to 71) <= SBOX2SubBytes(instate(8)) xor
SBOX3SubBytes(instate(13));
        pre1_mid_round_out(72 to 79) <= SBOXSubBytes(instate(8)) xor
SBOX2SubBytes(instate(13));
        pre1_mid_round_out(80 to 87) <= SBOXSubBytes(instate(8)) xor
SBOXSubBytes(instate(13));
        pre1_mid_round_out(88 to 95) <= SBOX3SubBytes(instate(8)) xor
SBOXSubBytes(instate(13));


        pre1_mid_round_out(96 to 103) <= SBOX2SubBytes(instate(12)) xor
SBOX3SubBytes(instate(1));
        pre1_mid_round_out(104 to 111) <= SBOXSubBytes(instate(12)) xor
SBOX2SubBytes(instate(1));
        pre1_mid_round_out(112 to 119) <= SBOXSubBytes(instate(12)) xor
SBOXSubBytes(instate(1));
        pre1_mid_round_out(120 to 127) <= SBOX3SubBytes(instate(12)) xor
SBOXSubBytes(instate(1));

        pre2_mid_round_out(0 to 7) <= SBOXSubBytes(instate(10)) xor
SBOXSubBytes(instate(15));
        pre2_mid_round_out(8 to 15) <= SBOX3SubBytes(instate(10)) xor
SBOXSubBytes(instate(15));
        pre2_mid_round_out(16 to 23) <= SBOX2SubBytes(instate(10)) xor
SBOX3SubBytes(instate(15));
        pre2_mid_round_out(24 to 31) <= SBOXSubBytes(instate(10)) xor
SBOX2SubBytes(instate(15));


        pre2_mid_round_out(32 to 39) <= SBOXSubBytes(instate(14)) xor
SBOXSubBytes(instate(3));
        pre2_mid_round_out(40 to 47) <= SBOX3SubBytes(instate(14)) xor
SBOXSubBytes(instate(3));
        pre2_mid_round_out(48 to 55) <= SBOX2SubBytes(instate(14)) xor
SBOX3SubBytes(instate(3));
        pre2_mid_round_out(56 to 63) <= SBOXSubBytes(instate(14)) xor
SBOX2SubBytes(instate(3));


        pre2_mid_round_out(64 to 71) <= SBOXSubBytes(instate(2)) xor
SBOXSubBytes(instate(7));
        pre2_mid_round_out(72 to 79) <= SBOX3SubBytes(instate(2)) xor
SBOXSubBytes(instate(7));
```

```
        pre2_mid_round_out(80 to 87) <= SBOX2SubBytes(instate(2)) xor
SBOX3SubBytes(instate(7));
        pre2_mid_round_out(88 to 95) <= SBOXSubBytes(instate(2)) xor
SBOX2SubBytes(instate(7));


        pre2_mid_round_out(96 to 103) <= SBOXSubBytes(instate(6)) xor
SBOXSubBytes(instate(11));
        pre2_mid_round_out(104 to 111) <= SBOX3SubBytes(instate(6)) xor
SBOXSubBytes(instate(11));
        pre2_mid_round_out(112 to 119) <= SBOX2SubBytes(instate(6)) xor
SBOX3SubBytes(instate(11));
        pre2_mid_round_out(120 to 127) <= SBOXSubBytes(instate(6)) xor
SBOX2SubBytes(instate(11));

        -----
        mid_round_out(0 to 7) <= pre1_mid_round_out(0 to 7) xor pre2_mid_round_out(0 to 7);
        mid_round_out(8 to 15) <= pre1_mid_round_out(8 to 15) xor pre2_mid_round_out(8 to
15);
        mid_round_out(16 to 23) <= pre1_mid_round_out(16 to 23) xor pre2_mid_round_out(16
to 23);
        mid_round_out(24 to 31) <= pre1_mid_round_out(24 to 31) xor pre2_mid_round_out(24
to 31);


        mid_round_out(32 to 39) <= pre1_mid_round_out(32 to 39) xor pre2_mid_round_out(32
to 39);
        mid_round_out(40 to 47) <= pre1_mid_round_out(40 to 47) xor pre2_mid_round_out(40
to 47);
        mid_round_out(48 to 55) <= pre1_mid_round_out(48 to 55) xor pre2_mid_round_out(48
to 55);
        mid_round_out(56 to 63) <= pre1_mid_round_out(56 to 63) xor pre2_mid_round_out(56
to 63);


        mid_round_out(64 to 71) <= pre1_mid_round_out(64 to 71) xor pre2_mid_round_out(64
to 71);
        mid_round_out(72 to 79) <= pre1_mid_round_out(72 to 79) xor pre2_mid_round_out(72
to 79);
        mid_round_out(80 to 87) <= pre1_mid_round_out(80 to 87) xor pre2_mid_round_out(80
to 87);
        mid_round_out(88 to 95) <= pre1_mid_round_out(88 to 95) xor pre2_mid_round_out(88
to 95);


        mid_round_out(96 to 103) <= pre1_mid_round_out(96 to 103) xor
pre2_mid_round_out(96 to 103);
        mid_round_out(104 to 111) <= pre1_mid_round_out(104 to 111) xor
pre2_mid_round_out(104 to 111);
        mid_round_out(112 to 119) <= pre1_mid_round_out(112 to 119) xor
pre2_mid_round_out(112 to 119);
        mid_round_out(120 to 127) <= pre1_mid_round_out(120 to 127) xor
pre2_mid_round_out(120 to 127);


        final_round_out(0 to 7) <= SBOXSubBytes(instate(0));
```

```
final_round_out(8 to 15) <= SBOXSubBytes(instate(5));
final_round_out(16 to 23) <= SBOXSubBytes(instate(10));
final_round_out(24 to 31) <= SBOXSubBytes(instate(15));

final_round_out(32 to 39) <= SBOXSubBytes(instate(4));
final_round_out(40 to 47) <= SBOXSubBytes(instate(9));
final_round_out(48 to 55) <= SBOXSubBytes(instate(14));
final_round_out(56 to 63) <= SBOXSubBytes(instate(3));

final_round_out(64 to 71) <= SBOXSubBytes(instate(8));
final_round_out(72 to 79) <= SBOXSubBytes(instate(13));
final_round_out(80 to 87) <= SBOXSubBytes(instate(2));
final_round_out(88 to 95) <= SBOXSubBytes(instate(7));

final_round_out(96 to 103) <= SBOXSubBytes(instate(12));
final_round_out(104 to 111) <= SBOXSubBytes(instate(1));
final_round_out(112 to 119) <= SBOXSubBytes(instate(6));
final_round_out(120 to 127) <= SBOXSubBytes(instate(11));

process (clock)
begin
 if (clock'event and clock = '1') then
        if (round_num_R = "1011") then
            aes_data_out_last <= final_round_out xor aes_key_in_R;
        end if;
 end if;
end process;

aes_data_out_round0 <= aes_data_in xor aes_key_in_R;
aes_data_out_mid <= mid_round_out xor aes_key_in_R;
aes_data_out_final <= final_round_out xor aes_key_in_R;

end rtl;
```

CONTROL SM

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity control_sm is
  Port (
                    -- i/f to input fifo
            data_in : in std_logic_vector(0 to 127);
    iv : in std_logic_vector(0 to 127);
    context : in std_logic_vector(0 to 31);
    fifo_emptyb : in std_logic;
    rdb_fifo : out std_logic;

                    -- i/f to output fifo
    wrb_fifo : out std_logic;
    fifo_fullb : in std_logic;

                    -- i/f to cipher block
    aes_data_in : out std_logic_vector(0 to 127);
```

```
                round : out std_logic_vector(0 to 3);
                aes_data_out_round0 : in std_logic_vector(0 to 127);
                        aes_data_out_mid : in std_logic_vector(0 to 127);
                    aes_data_out_final : in std_logic_vector(0 to 127);
                aes_data_out_last : in std_logic_vector(0 to 127);

                                -- i/f to key memory
                aes_key_mem_address : out std_logic_vector(0 to 4);
                read_key_mem : out std_logic;

                                clock : in std_logic;
                                reset : in std_logic
                                );
        end control_sm;

        architecture RTL of control_sm is

        type state_type is (IDLE, IDLEOUT, Get_Context, Get_ContextOut, CBC_Encrypt_11,
ECB_Encrypt_11, ECB_Encrypt_1, ECB_Encrypt_2, ECB_Encrypt_3, ECB_Encrypt_4,
ECB_Encrypt_5, ECB_Encrypt_6, ECB_Encrypt_7, ECB_Encrypt_8, ECB_Encrypt_9,
ECB_Encrypt_10, CBC_Encrypt_1, CBC_Encrypt_2, CBC_Encrypt_3, CBC_Encrypt_4,
CBC_Encrypt_5, CBC_Encrypt_6, CBC_Encrypt_7, CBC_Encrypt_8, CBC_Encrypt_9,
CBC_Encrypt_10, Out_Fifo_Full);
        signal ST, nST : state_type;
        signal sm_cipher_key_index: std_logic;
        signal cipher_input: std_logic_vector(0 to 127);
        signal sm_context_input: std_logic_vector (0 to 31);
        signal sm_iv_input: std_logic_vector (0 to 127);
        signal cclock: std_logic;

        begin

        sm_context_input(1) <= context(1);

        cclock <= clock;
        cipher_controller: process (clock)
        begin
                if (clock'event and clock = '1') then
                case ST is
                        when IDLE =>
                                if (fifo_emptyb = '1') then
                                        rdb_fifo <= '0';
                                        read_key_mem <= '0';
                                        round <= "0000";
                                        aes_data_in <=
"00000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000";
                                        aes_key_mem_address <= "00000";
                                        wrb_fifo <= '1';
                                        cipher_input <= data_in;
                                        sm_context_input <= context;
                                        sm_iv_input <= iv;
                                        --sm_cipher_key_index <= '0';
                                        ST <= Get_Context;
                                else
                                        rdb_fifo <= '1';
```

```
                                        read_key_mem <= '0';
                                        round <= "0000";
                                        aes_data_in <=
"00000000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000";
                                        aes_key_mem_address <= "00000";
                                        wrb_fifo <= '1';
                                        ST <= IDLE;
                              end if;

                    when IDLEOUT =>
                              if (fifo_emptyb = '1') then
                                        rdb_fifo <= '0';
                                        read_key_mem <= '0';
                                        round <= "0000";
                                        --cipher_output <= aes_data_out_last;
                                        aes_data_in <=
"00000000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000";
                                        aes_key_mem_address <= "00000";
                                        wrb_fifo <= '0';
                                        cipher_input <= data_in;
                                        sm_context_input <= context;
                                        sm_iv_input <= iv;
                                        ST <= Get_Context;
                              else
                                        rdb_fifo <= '1';
                                        read_key_mem <= '0';
                                        round <= "0000";
                                        --cipher_output <= aes_data_out_last;
                                        aes_data_in <=
"00000000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000";
                                        aes_key_mem_address <= "00000";
                                        wrb_fifo <= '0';
                                        ST <= IDLE;
                              end if;

                    when Get_Context =>
                              if (sm_context_input(2 to 3) = "01") then  --- ECB Mode
                                        rdb_fifo <= '1';
                                        read_key_mem <= '1';
                                        round <= "0000";
                                        aes_data_in <=
"00000000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000";
                                        sm_cipher_key_index <= sm_context_input(4);
                                        aes_key_mem_address <= sm_context_input(4) &
"0000";

                                        wrb_fifo <= '1';
                                        ST <= ECB_Encrypt_1;

                              elsif (sm_context_input(2 to 3) = "10") then ---CBC mode
                                        rdb_fifo <= '1';
                                        read_key_mem <= '1';
                                        round <= "0000";
```

```
                                        aes_data_in <=
"00000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000";
                                        sm_cipher_key_index <= sm_context_input(4);
                                        aes_key_mem_address <= sm_context_input(4) &
"0000";
                                        wrb_fifo <= '1';
                                        ST <= CBC_Encrypt_1;

                                else
                                        rdb_fifo <= '1';                    --- undefined mode
                                        read_key_mem <= '0';
                                        round <= "0000";
                                        aes_data_in <=
"00000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000";
                                        aes_key_mem_address <= "00000";
                                        wrb_fifo <= '1';
                                        ST <= IDLE;
                                end if;
                        when Get_ContextOut =>
                                if (sm_context_input(2 to 3) = "01") then  --- ECB Mode
                                        rdb_fifo <= '1';
                                        read_key_mem <= '1';
                                        round <= "0000";
                                        --cipher_output <= aes_data_out_last;
                                        aes_data_in <=
"00000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000";
                                        sm_cipher_key_index <= sm_context_input(4);
                                        aes_key_mem_address <= sm_context_input(4) &
"0000";
                                        wrb_fifo <= '1';
                                        ST <= ECB_Encrypt_1;

                                elsif (sm_context_input(2 to 3) = "10") then ---CBC mode
                                        rdb_fifo <= '1';
                                        read_key_mem <= '1';
                                        round <= "0000";
                                        --cipher_output <= aes_data_out_last;
                                        aes_data_in <=
"00000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000";
                                        sm_cipher_key_index <= sm_context_input(4);
                                        aes_key_mem_address <= sm_context_input(4) &
"0000";
                                        wrb_fifo <= '1';
                                        ST <= CBC_Encrypt_1;

                                else
                                        rdb_fifo <= '1';                    --- undefined mode
                                        read_key_mem <= '0';
                                        round <= "0000";
                                        --cipher_output <= aes_data_out_last;
```

```
                                        aes_data_in <=
"00000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000";
                                        aes_key_mem_address <= "00000";
                                        wrb_fifo <= '1';
                                        ST <= IDLE;
                                end if;


                        when ECB_Encrypt_11 =>
                                if ((fifo_emptyb = '1') and (fifo_fullb = '1')) then  --- ECB Mode
                                        rdb_fifo <= '0';
                                        read_key_mem <= '1';
                                        round <= "1011";
                                        aes_data_in <= aes_data_out_mid;
                                        aes_key_mem_address <= sm_cipher_key_index &
"1010";

                                        wrb_fifo <= '0';
                                        cipher_input <= data_in;
                                        sm_context_input <= context;
                                        sm_iv_input <= iv;
                                        ST <= Get_ContextOut;

                                elsif ((fifo_emptyb = '1') and (fifo_fullb = '0')) then  --- ECB Mode
                                        rdb_fifo <= '1';
                                        read_key_mem <= '0';
                                        round <= "1011";
                                        aes_data_in <= aes_data_out_mid;
                                        aes_key_mem_address <= sm_cipher_key_index &
"1010";

                                        wrb_fifo <= '1';
                                        ST <= Out_Fifo_Full;

                                elsif ((fifo_emptyb = '0') and (fifo_fullb = '0')) then  --- ECB Mode
                                        rdb_fifo <= '1';
                                        read_key_mem <= '0';
                                        round <= "1011";
                                        aes_data_in <= aes_data_out_mid;
                                        aes_key_mem_address <= sm_cipher_key_index &
"1010";

                                        wrb_fifo <= '1';
                                        ST <= Out_Fifo_Full;

                                elsif ((fifo_emptyb = '0') and (fifo_fullb = '1')) then  --- ECB Mode
                                        rdb_fifo <= '0';
                                        read_key_mem <= '1';
                                        round <= "1011";
                                        aes_data_in <= aes_data_out_mid;
                                        aes_key_mem_address <= sm_cipher_key_index &
"1010";

                                        wrb_fifo <= '0';
                                        ST <= IDLEOUT;

                                else
                                        rdb_fifo <= '1';                        --- undefined mode
                                        read_key_mem <= '0';
```

```
                                        round <= "0000";
                                        --cipher_output <= aes_data_out_mid;
                                        aes_data_in <=
"000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000";
                                        aes_key_mem_address <= "00000";
                                        wrb_fifo <= '1';
                                        ST <= IDLE;

                                end if;

                        when ECB_Encrypt_1 =>   --- ECB Mode
                                        rdb_fifo <= '1';
                                        read_key_mem <= '1';
                                        round <= "0001";
                                        aes_data_in <= cipher_input;
                                        aes_key_mem_address <= sm_cipher_key_index &
"0000";
                                        wrb_fifo <= '1';
                                        ST <= ECB_Encrypt_2;

                        when ECB_Encrypt_2 =>
                                        rdb_fifo <= '1';
                                        read_key_mem <= '1';
                                        round <= "0010";
                                        aes_data_in <= aes_data_out_round0;
                                        aes_key_mem_address <= sm_cipher_key_index &
"0001";
                                        wrb_fifo <= '1';
                                        ST <= ECB_Encrypt_3;

                        when ECB_Encrypt_3 =>
                                        rdb_fifo <= '1';
                                        read_key_mem <= '1';
                                        round <= "0011";
                                        aes_data_in <= aes_data_out_mid;
                                        aes_key_mem_address <= sm_cipher_key_index &
"0010";
                                        wrb_fifo <= '1';
                                        ST <= ECB_Encrypt_4;

                        when ECB_Encrypt_4   =>
                                        rdb_fifo <= '1';
                                        read_key_mem <= '1';
                                        round <= "0100";
                                        aes_data_in <= aes_data_out_mid;
                                        aes_key_mem_address <= sm_cipher_key_index &
"0011";
                                        wrb_fifo <= '1';
                                        ST <= ECB_Encrypt_5;

                        when ECB_Encrypt_5  =>
                                        rdb_fifo <= '1';
                                        read_key_mem <= '1';
                                        round <= "0101";
                                        aes_data_in <= aes_data_out_mid;
```

```
                        aes_key_mem_address <= sm_cipher_key_index &
"0100";
                        wrb_fifo <= '1';
                        ST <= ECB_Encrypt_6;

            when ECB_Encrypt_6  =>
                        rdb_fifo <= '1';
                        read_key_mem <= '1';
                        round <= "0110";
                        aes_data_in <= aes_data_out_mid;
                        aes_key_mem_address <= sm_cipher_key_index &
"0101";
                        wrb_fifo <= '1';
                        ST <= ECB_Encrypt_7;

            when ECB_Encrypt_7  =>
                        rdb_fifo <= '1';
                        read_key_mem <= '1';
                        round <= "0111";
                        aes_data_in <= aes_data_out_mid;
                        aes_key_mem_address <= sm_cipher_key_index &
"0110";
                        wrb_fifo <= '1';
                        ST <= ECB_Encrypt_8;

            when ECB_Encrypt_8 =>
                        rdb_fifo <= '1';
                        read_key_mem <= '1';
                        round <= "1000";
                        aes_data_in <= aes_data_out_mid;
                        aes_key_mem_address <= sm_cipher_key_index &
"0111";
                        wrb_fifo <= '1';
                        ST <= ECB_Encrypt_9;

            when ECB_Encrypt_9  =>
                        rdb_fifo <= '1';
                        read_key_mem <= '1';
                        round <= "1001";
                        aes_data_in <= aes_data_out_mid;
                        aes_key_mem_address <= sm_cipher_key_index &
"1000";
                        wrb_fifo <= '1';
                        ST <= ECB_Encrypt_10;

            when ECB_Encrypt_10  =>
                        rdb_fifo <= '1';
                        read_key_mem <= '1';
                        round <= "1010";
                        aes_data_in <= aes_data_out_mid;
                        aes_key_mem_address <= sm_cipher_key_index &
"1001";
                        wrb_fifo <= '1';
                        ST <= ECB_Encrypt_11;

            when CBC_Encrypt_11 =>
```

```
                    if ((fifo_emptyb = '1') and (fifo_fullb = '1')) then  --- ECB Mode
                              rdb_fifo <= '0';
                              read_key_mem <= '1';
                              round <= "1011";
                              aes_data_in <= aes_data_out_mid;
                              aes_key_mem_address <= sm_cipher_key_index &
"1010";

                              wrb_fifo <= '0';
                              cipher_input <= data_in;
                              sm_context_input <= context;
                              sm_iv_input <= iv;
                              ST <= Get_ContextOut;

                    elsif ((fifo_emptyb = '1') and (fifo_fullb = '0')) then  --- ECB Mode
                              rdb_fifo <= '1';
                              read_key_mem <= '0';
                              round <= "1011";
                              aes_data_in <= aes_data_out_mid;
                              aes_key_mem_address <= sm_cipher_key_index &
"1010";

                              wrb_fifo <= '1';
                              ST <= Out_Fifo_Full;

                    elsif ((fifo_emptyb = '0') and (fifo_fullb = '0')) then  --- ECB Mode
                              rdb_fifo <= '1';
                              read_key_mem <= '0';
                              round <= "1011";
                              aes_data_in <= aes_data_out_mid;
                              aes_key_mem_address <= sm_cipher_key_index &
"1010";

                              wrb_fifo <= '1';
                              ST <= Out_Fifo_Full;

                    elsif ((fifo_emptyb = '0') and (fifo_fullb = '1')) then  --- ECB Mode
                              rdb_fifo <= '0';
                              read_key_mem <= '1';
                              round <= "1011";
                              aes_data_in <= aes_data_out_mid;
                              aes_key_mem_address <= sm_cipher_key_index &
"1010";

                              wrb_fifo <= '0';
                              ST <= IDLEOUT;

                    else
                              rdb_fifo <= '1';                    --- undefined mode
                              read_key_mem <= '0';
                              round <= "0000";
                              --cipher_output <= aes_data_out_mid;
                              aes_data_in <=
"00000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000";
                              aes_key_mem_address <= "00000";
                              wrb_fifo <= '1';
                              ST <= IDLE;

                    end if;
```

```
when CBC_Encrypt_1 =>  --- CBC Mode
        rdb_fifo <= '1';
        read_key_mem <= '1';
        round <= "0001";
        if (sm_context_input(0) = '1') then -- SOP = 1
                aes_data_in <= cipher_input xor sm_iv_input;
        elsif (sm_context_input(0) = '0') then -- middle or end of
packet
                aes_data_in <= cipher_input xor
aes_data_out_last;
        end if;
        aes_key_mem_address <= sm_cipher_key_index &
"0000";
        wrb_fifo <= '1';
        ST <= CBC_Encrypt_2;

when CBC_Encrypt_2 =>
        rdb_fifo <= '1';
        read_key_mem <= '1';
        round <= "0010";
        aes_data_in <= aes_data_out_round0;
        aes_key_mem_address <= sm_cipher_key_index &
"0001";
        wrb_fifo <= '1';
        ST <= CBC_Encrypt_3;

when CBC_Encrypt_3 =>
        rdb_fifo <= '1';
        read_key_mem <= '1';
        round <= "0011";
        aes_data_in <= aes_data_out_mid;
        aes_key_mem_address <= sm_cipher_key_index &
"0010";
        wrb_fifo <= '1';
        ST <= CBC_Encrypt_4;

when CBC_Encrypt_4 =>
        rdb_fifo <= '1';
        read_key_mem <= '1';
        round <= "0100";
        aes_data_in <= aes_data_out_mid;
        aes_key_mem_address <= sm_cipher_key_index &
"0011";
        wrb_fifo <= '1';
        ST <= CBC_Encrypt_5;

when CBC_Encrypt_5 =>
        rdb_fifo <= '1';
        read_key_mem <= '1';
        round <= "0101";
        aes_data_in <= aes_data_out_mid;
        aes_key_mem_address <= sm_cipher_key_index &
"0100";
        wrb_fifo <= '1';
        ST <= CBC_Encrypt_6;
```

```
when CBC_Encrypt_6 =>
        rdb_fifo <= '1';
        read_key_mem <= '1';
        round <= "0110";
        aes_data_in <= aes_data_out_mid;
        aes_key_mem_address <= sm_cipher_key_index &
"0101";

        wrb_fifo <= '1';
        ST <= CBC_Encrypt_7;


when CBC_Encrypt_7 =>
        rdb_fifo <= '1';
        read_key_mem <= '1';
        round <= "0111";
        aes_data_in <= aes_data_out_mid;
        aes_key_mem_address <= sm_cipher_key_index &
"0110";

        wrb_fifo <= '1';
        ST <= CBC_Encrypt_8;

when CBC_Encrypt_8 =>
        rdb_fifo <= '1';
        read_key_mem <= '1';
        round <= "1000";
        aes_data_in <= aes_data_out_mid;
        aes_key_mem_address <= sm_cipher_key_index &
"0111";

        wrb_fifo <= '1';
        ST <= CBC_Encrypt_9;

when CBC_Encrypt_9 =>
        rdb_fifo <= '1';
        read_key_mem <= '1';
        round <= "1001";
        aes_data_in <= aes_data_out_mid;
        aes_key_mem_address <= sm_cipher_key_index &
"1000";

        wrb_fifo <= '1';
        ST <= CBC_Encrypt_10;

when CBC_Encrypt_10 =>
        rdb_fifo <= '1';
        read_key_mem <= '1';
        round <= "1010";
        aes_data_in <= aes_data_out_mid;
        aes_key_mem_address <= sm_cipher_key_index &
"1001";

        wrb_fifo <= '1';
        ST <= CBC_Encrypt_11;

when Out_Fifo_Full =>
    if ((fifo_emptyb = '1') and (fifo_fullb = '1')) then  --input FIFO not
empty, output FIFO not FULL
        rdb_fifo <= '0';
```

```
                                        read_key_mem <= '1';
                                        round <= "1100";
                                        aes_data_in <= aes_data_out_last;
                                        aes_key_mem_address <= sm_cipher_key_index &
"1011";
                                        wrb_fifo <= '0';
                                        cipher_input <= data_in;
                                        sm_context_input <= context;
                                        sm_iv_input <= iv;
                                        ST <= Get_ContextOut;

                                elsif ((fifo_emptyb = '0') and (fifo_fullb = '1')) then  --input FIFO
empty, output FIFO not FULL
                                        rdb_fifo <= '0';
                                        read_key_mem <= '1';
                                        round <= "1100";
                                        aes_data_in <= aes_data_out_last;
                                        aes_key_mem_address <= sm_cipher_key_index &
"1011";
                                        wrb_fifo <= '0';
                                        ST <= IDLEOUT;

                                else
                                        rdb_fifo <= '1';
                                        read_key_mem <= '1';
                                        round <= "1100";
                                        aes_data_in <= aes_data_out_last;
                                        aes_key_mem_address <= sm_cipher_key_index &
"1011";
                                        wrb_fifo <= '1';
                                        ST <= Out_Fifo_Full;
                                end if;


                        when others =>
                                ST <= IDLE;

                        end case;
                        end if;
        end process;

end RTL;
```

INPUT FIFO

```
        library IEEE;
        use IEEE.STD_LOGIC_1164.ALL;
        use IEEE.STD_LOGIC_ARITH.ALL;
        use IEEE.STD_LOGIC_UNSIGNED.ALL;

        entity FIFO is
          Port ( resetb : in std_logic;
                clock : in std_logic;
                rdb : in std_logic;
                wrb : in std_logic;
                data_in : in std_logic_vector(0 to 63);
```

```
                                    iv_in : in std_logic_vector (0 to 63);
                                    context_in : in std_logic_vector (0 to 15);
          emptyb : out std_logic;
          fullb : out std_logic;
                                    context_out : out std_logic_vector (0 to 31);
                                    iv_out : out std_logic_vector (0 to 127);
          data_out : out std_logic_vector(0 to 127));
end FIFO;

architecture RTL of FIFO is
signal read_ptr: std_logic_vector(0 to 2);
signal write_ptr: std_logic_vector (0 to 2);
type data_registerType is array (0 to 7) of std_logic_vector(0 to 63);
signal data_register: data_registerType;
type context_registerType is array (0 to 7) of std_logic_vector(0 to 15);
signal context_register: context_registerType;
type iv_registerType is array (0 to 7) of std_logic_vector(0 to 63);
signal iv_register: iv_registerType;
signal write_read: std_logic_vector (0 to 1);
signal fullb_flag: std_logic;
signal emptyb_flag: std_logic;
SIGNAL contents_counter: INTEGER range 0 to 16;

begin
inp_latch: process (clock)
begin
          if (clock'event and clock = '1') then
--                  if (resetb = '0') then
--                            write_read <= "11";
--                  else
                              write_read <= wrb & rdb;
--                  end if;
          end if;
end process;

fifo: process (clock)
begin
          if (clock'event and clock = '1') then
             if (resetb = '0') then
                              write_ptr <= "000";
                              read_ptr <= "000";
                              fullb_flag <= '1';
                              emptyb_flag <= '0';
                              contents_counter <= 0;
                              for i in 0 to 7 loop
                                        data_register(i) <=
"0000000000000000000000000000000000000000000000000000000000000000";
                                        iv_register(i) <=
"0000000000000000000000000000000000000000000000000000000000000000";
                                        context_register(i) <= "0000000000000000";
                              end loop;
                    else

                              case write_read is
                              when "00" =>
                                        read_ptr <= read_ptr + 2;
```

```
                                        data_register(conv_integer(write_ptr)) <= data_in;
                                        iv_register(conv_integer(write_ptr)) <= iv_in;
                                        context_register(conv_integer(write_ptr)) <= context_in;
                                        write_ptr <= write_ptr + 1;
                                        contents_counter <= contents_counter - 1;
                                   when "01" =>
                                     contents_counter <= contents_counter + 1;
                                   if (contents_counter > 0) then
                                             emptyb_flag <= '1';
                                        else
                                             emptyb_flag <= '0';
                                        end if;
                                        if (fullb_flag = '1') then
                                             data_register(conv_integer(write_ptr)) <=
data_in;
                                             iv_register(conv_integer(write_ptr)) <= iv_in;
                                             context_register(conv_integer(write_ptr)) <=
context_in;
                                             if (write_ptr + 2) = read_ptr then
                                                  fullb_flag <= '0';
                                             end if;
                                             write_ptr <= write_ptr + 1;
                                        end if;
                                   when "10" =>
                                        if (emptyb_flag = '1') then
                                             if (read_ptr + 2) = write_ptr then
                                                  emptyb_flag <= '0';
                                             end if;
                                             read_ptr <= read_ptr + 2;
                                             contents_counter <= contents_counter - 2;
                                        end if;
                                        fullb_flag <= '1';
                                   when others => null;
                                   end case;
                         end if;
                    end if;
          end process;
          fullb <= fullb_flag;

          output: process (clock)
          begin
                    if  (clock'event and clock = '1') then
                              data_out <= data_register(conv_integer(read_ptr)) &
data_register(conv_integer(read_ptr+1));
                              iv_out <= iv_register(conv_integer(read_ptr)) &
iv_register(conv_integer(read_ptr+1));
                              context_out <= context_register(conv_integer(read_ptr)) &
context_register(conv_integer(read_ptr+1));
                              emptyb <= emptyb_flag;
                    end if;
          end process;
end RTL;
```

OUTPUT FIFO

```
          library IEEE;
```

```
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity OUT_FIFO is
   Port ( resetb : in std_logic;
          clock : in std_logic;
          rdb : in std_logic;
          wrb : in std_logic;
          data_in : in std_logic_vector(0 to 127);
          emptyb : out std_logic;
          fullb : out std_logic;
          data_out : out std_logic_vector(0 to 63));
end OUT_FIFO;

architecture RTL of OUT_FIFO is
signal read_ptr: std_logic_vector(0 to 2);
signal write_ptr: std_logic_vector (0 to 2);
type data_registerType is array (0 to 7) of std_logic_vector(0 to 63);
signal data_register: data_registerType;
signal write_read: std_logic_vector (0 to 1);
signal fullb_flag: std_logic;
signal emptyb_flag: std_logic;
SIGNAL contents_counter: INTEGER range 0 to 16;

begin

outp_latch: process (clock)
begin
         if (clock'event and clock = '1') then
--                if (resetb = '0') then
--                       write_read <= "11";
--                else
                         write_read <= wrb & rdb;
--                end if;
         end if;
end process;

fifo: process (clock)
begin

         if (clock'event and clock = '1') then
                if (resetb = '0') then
                       write_ptr <= "000";
                       read_ptr <= "000";
                       --data_out <=
"0000000000000000000000000000000000000000000000000000000000000000";
                       fullb_flag <= '1';
                       emptyb_flag <= '0';
                       contents_counter <= 0;
                       for i in 0 to 7 loop
                              data_register(i) <=
"0000000000000000000000000000000000000000000000000000000000000000";
                       end loop;
                else
                case write_read is
```

```
                                when "00" =>
                                        data_out <= data_register(conv_integer(read_ptr));
                                        read_ptr <= read_ptr + 1;
                                        data_register(conv_integer(write_ptr)) <= data_in(0 to
63);
                                        data_register(conv_integer(write_ptr + 1)) <= data_in(64
to 127);
                                        write_ptr <= write_ptr + 2;
                                        contents_counter <= contents_counter + 1;
                                when "01" =>
                                        emptyb_flag <= '1';
                                        if (fullb_flag = '1') then
                                                data_register(conv_integer(write_ptr)) <=
data_in(0 to 63);
                                                data_register(conv_integer(write_ptr+1)) <=
data_in(64 to 127);
                                                write_ptr <= write_ptr + 2;
                                                contents_counter <= contents_counter + 2;
                                                if (write_ptr = read_ptr) then
                                                        if (contents_counter > 6) then
                                                                fullb_flag <= '0';
                                                        else
                                                                fullb_flag <= '1';
                                                        end if;
                                                end if;

                                        end if;
                                when "10" =>
                                        if (emptyb_flag = '1') then
                                                if (read_ptr + 1) = write_ptr then
                                                        emptyb_flag <= '0';
                                                end if;
                                                data_out <=
data_register(conv_integer(read_ptr));
                                                read_ptr <= read_ptr + 1;
                                                contents_counter <= contents_counter - 1;
                                        end if;

                                        fullb_flag <= '1';
                                when others => null;
                        end case;
                        end if;
                end if;
        end process;

        emptyb <= emptyb_flag;

        outfif: process (clock)
        begin
                if  (clock'event and clock = '1') then
                fullb <= fullb_flag;

                end if;
        end process;
end RTL;
```

AES_MODULE_TB

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.numeric_std.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

ENTITY testbench IS
END testbench;

ARCHITECTURE behavior OF testbench IS

        COMPONENT aes_cipher_module_3
        PORT(
                data_input : IN std_logic_vector(0 to 63);
                iv_in : IN std_logic_vector(0 to 63);
                context_in : IN std_logic_vector(0 to 15);
                wrb : IN std_logic;
                rdb : IN std_logic;
                key_in : IN std_logic_vector(0 to 127);
                clock : IN std_logic;
                reset : IN std_logic;
                fullb : OUT std_logic;
                data_output : OUT std_logic_vector(0 to 63);
                emptyb : OUT std_logic;
                key_address : OUT std_logic_vector(0 to 4);
                read_mem : OUT std_logic
                );
        END COMPONENT;

        SIGNAL data_input :  std_logic_vector(0 to 63) :=
"1110111011101110111011101110111011101110111011101110111011101110";
        SIGNAL iv_in :  std_logic_vector(0 to 63) :=
"1110111011101110111011101110111011101110111011101110111011101110";
        SIGNAL context_in :  std_logic_vector(0 to 15) := "0111011101110111";
        SIGNAL wrb :  std_logic := '1';
        SIGNAL fullb :  std_logic;
        SIGNAL data_output :  std_logic_vector(0 to 63);
        SIGNAL emptyb :  std_logic;
        SIGNAL rdb :  std_logic := '1';
        SIGNAL key_in :  std_logic_vector(0 to 127);
        SIGNAL key_address :  std_logic_vector(0 to 4);
        SIGNAL read_mem :  std_logic;
        SIGNAL clock :  std_logic := '0';
        SIGNAL reset :  std_logic := '0';
--      type KeyMemoryType is array (0 to 31) of std_logic_vector(0 to 127);
-- signal KeyMemory : KeyMemoryType;
--      SIGNAL rdy_counter: INTEGER range 0 to 16;
        SIGNAL rdy_counter: INTEGER :=0;

subtype KEY_BOX_FIELD      is std_logic_vector (0 to 127);
subtype KEYBOX_INDEX_TYPE  is std_logic_vector (0 to 4);
type KEY_BOX_TYPE      is array (0 to 31) of KEY_BOX_FIELD;
constant KEY_BOX : KEY_BOX_TYPE := (
```

"1100001010000110011010010110110110001000011111001001101010100000011000
0100011011101110110011111000100000001001011010010001011010",
"1111110011001111110101111101101001110100101100110100110101111010000101
0110101000111101100100010000110101100011010101001000011110",
"1010001111001111101001010100110011010111011111001110100000110110110000
1011010100000111100111001011110111010110010100110001101100",
"0110110011100110111101010010010010111011100110100001110100010010011110
0101001110000000110110000010001110000101110100111100001100",
"1001010001100010000010110011110100101111111110000001011000101111010101
1010110110000101010100111111011000101000010101101001000011",
"1011011011011100000100010101110010011001001001000000011101110011110011
1110010010000100100011110000010111001100110100100001111111",
"0101010110001110110000111010110011001100101010101100010011011111000000
1100111000110101101110001100010100000010111001111010011100",
"0011111010000101000111010101011011110010001011111101100110001001111100
0100010111000011110110101011100101000111001001000111110110",
"0010001000000100010111111000111111010000001010111000011000000110001000
0100111100100010010110110011000100001000000001100010011010",
"1000111010101001111001111001001101011110100000100110000110010101011111
1110111110111010001111100110111011100111101111000001100011",
"10110011001001010001110001111001111011011010011101111110111101100100100
1000011001100101010001010100101001100001110110010101110110",
"0001001100010001000111010111111111110001110010100010010100001011111100
1100000111101001111000101101001101001010110011000011000101",
"0000000000000001000000100000001100000100000001010000011000000111000010
0000001001000010100000101100001100000011010000111000001111",
"0000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000",
"0000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000",
"0000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000",

"0000000000000001000000100000001100000100000001010000011000000111000010
0000001001000010100000101100001100000011010000111000001111",
"1101011010101010011101001111101110100101010111011100101111010110110
1010100110011110001111000111010110101010110111011011111110",
"1011011010010010110011110000101101100100001111011011110111110001101111
1010011011110001010000000001101000001100001011001111111110",
"1011011011111111011101000100111011010010110000101100100110111111011011
0001011001000011001011111100000100011010011011111101000001",
"0100011111110111111101111011110010010101001101010011111000000011111110
0101101100001100101011110011111101000001011000110111111101",
"0011110010101010101000111110100010101001100111111001110111101011010100
0011110011101011110101011110101101111101100010001010101010",
"010111100011100100001111011111101111110111101001101001001010010110101001
1101010101001111011100000100001010101000110001111101101011",
"00010100111110010111000000011010111000110101111111110001010001100010001
0000001010110111111010011010100111010101001110000000100110",
"0100011101000011100001110011010110100100000111000110010110111001111000
0000010110101110101111010010101110101111110111101011010010",
"0101010010011001001100101101000111110000100001010101011101101000000100
0010010011111011011001110010111110001011001001011101001110",
"0001001100010001000111010111111111110001110010100010010100001011111100
1100000111101001111000101101001101001010110011000011000101",

```
        "0001001100010001000111010111111111100011100101000100101000010111111100
1100000111101001111000101101001101001010110011000011000101",
        "0000000000000001000000100000001100000100000001010000011000000111000010
0000001001000010100000010110000110000001101000011100000111 1",
        "0000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000",
        "0000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000",
        "0000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000"

        );

        BEGIN

                uut: aes_cipher_module_3 PORT MAP(
                        data_input => data_input,
                        iv_in => iv_in,
                        context_in => context_in,
                        wrb => wrb,
                        fullb => fullb,
                        data_output => data_output,
                        emptyb => emptyb,
                        rdb => rdb,
                        key_in => key_in,
                        key_address => key_address,
                        read_mem => read_mem,
                        clock => clock,
                        reset => reset
                );

        -- *** Test Bench - User Defined Section ***
          tb : PROCESS (clock)
          BEGIN
                        if (clock'event and clock = '1') then
                                if (rdy_counter = 0) then
                                        reset <= '0' after 1 ns;
                                        rdy_counter <= rdy_counter + 1;
                                elsif (rdy_counter = 1) then
                                        reset <= '1' after 1 ns;
                                        rdy_counter <= rdy_counter + 1;
                                elsif (rdy_counter = 2) then
                                        reset <= '1' after 1 ns;
                                        rdy_counter <= rdy_counter + 1;
                                        if (fullb = '1') then
                                                wrb <= '0' after 1 ns;
                                        end if;
                                elsif (rdy_counter = 3) then
                                        if (fullb = '1') then
                                                wrb <= '0' after 1 ns;   --ecb
                                            data_input <=
"0000000000010001001000100011001101000100010101010110011001110111" after 1 ns;
                                                iv_in     <=
"0000000000000000000000000000000000000000000000000000000000000000" after 1 ns;
                                                context_in <= "0001100000000000" after 1 ns;
                                                rdy_counter <= rdy_counter + 1 after 1 ns;
```

```
                    else
                            wrb <= '1' after 1 ns;
                    end if;
            elsif (rdy_counter = 4) then
                    if (fullb = '1') then
                            wrb <= '1' after 1 ns; --ecb, 2
                        data_input <=
"1000100010011001101010101011101111001100110111011110111011111111" after 1 ns;
                            iv_in     <=
"0000000000000000000000000000000000000000000000000000000000000001" after 1 ns;
                            context_in <= "0111100000000001" after 1 ns;
                            rdy_counter <= rdy_counter + 1 after 1 ns;
            else
                            wrb <= '1' after 1 ns;
                    end if;
            elsif (rdy_counter = 5) then
                    if (fullb = '1') then
                            wrb <= '0' after 1 ns; --nothing
                        data_input <=
"1000100010011001101010101011101111001100110111011110111011111111" after 1 ns;
                            iv_in     <=
"0000000000000000000000000000000000000000000000000000000000000001" after 1 ns;
                            context_in <= "0111100000000001" after 1 ns;
                            rdy_counter <= rdy_counter + 1 after 1 ns;
            else
                            wrb <= '1' after 1 ns;
                    end if;
            elsif (rdy_counter = 6) then
                    if (fullb = '1') then
                            wrb <= '0' after 1 ns; --cbc1, 1
                        data_input <=
"0000000000000001000000100000001100000100000001010000011000000111" after 1 ns;
                            iv_in     <=
"0101011000101110000101111001100101101101000010010011110100101000" after 1 ns;
                            context_in <= "1010000000000000" after 1 ns;
                            rdy_counter <= rdy_counter + 1 after 1 ns;
            else
                            wrb <= '1' after 1 ns;
                    end if;
            elsif (rdy_counter = 7) then
                    if (fullb = '1') then
                            wrb <= '0' after 1 ns; --cbc1 ,2
                        data_input <=
"0000100000001001000010100000101100001100000011010000111000001111" after 1 ns;
                            iv_in     <=
"1101110110110011101110100110100101011010001011100110111101011000" after 1 ns;
                            context_in <= "0010000000000001" after 1 ns;
                            rdy_counter <= rdy_counter + 1 after 1 ns;
            else
                            wrb <= '1' after 1 ns;
                    end if;
            elsif (rdy_counter = 8) then
                    if (fullb = '1') then
                            wrb <= '0' after 1 ns; --cbc2, 1
                        data_input <=
"0001000000010001000100100001001100010100000101010001011000010111" after 1 ns;
```

```
                                        iv_in      <=
"1101110110110011101111010011010010101101000101110011011111010110 00" after 1 ns;
                                        context_in <= "0010000000000001" after 1 ns;
                                        rdy_counter <= rdy_counter + 1 after 1 ns;
                        else
                                        wrb <= '1'after 1 ns;
                            end if;
                        elsif (rdy_counter = 9) then
                            if (fullb = '1') then
                                        wrb <= '1' after 1 ns; --cbc2, 2
                                        data_input <=
"0001100000011001000110100001101100011100000111010001111000011111" after 1 ns;
                                        iv_in      <=
"1101110110110011101111010011010010101101000101110011011111010110 00" after 1 ns;
                                        context_in <= "0010000000000001" after 1 ns;
                                        rdy_counter <= rdy_counter + 1 after 1 ns;
                            else
                                        wrb <= '1' after 1 ns;
                            end if;
                        else
                            if (fullb = '1') then
                                data_input <=
"1111000000000000000000000000000000000000000000000000000000000000" after 1 ns;
                                        iv_in      <=
"1100111100000000000000000000000000000000000000000000000000000011" after 1 ns;
                                        context_in <= "1111100000000011" after 1 ns;
                                        wrb <= '1' after 1 ns;
                                        rdy_counter <= rdy_counter + 1 after 1 ns;
                            else
                                        wrb <= '1' after 1 ns;
                            end if;
                        end if;
                    end if;
    END PROCESS;

    tb2 : PROCESS (clock)
    BEGIN
                    if (clock'event and clock = '1') then
                            if (emptyb = '1') then
                                        rdb <= '0' after 1 ns;
                            else
                                        rdb <= '1' after 1 ns;
                            end if;
                    end if;
    END PROCESS;

    with read_mem select
     key_in <= KEY_BOX(conv_integer(key_address)) when '1',

"000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000" when others;


END;
```

REFERENCES

[1] N. Doraswamy and D. Harkins, *IPSec.* Saddle River, NJ: Prentice Hall PTR, 1999.

[2] R. Venkateswaran, "Virtual Private Networks," *IEEE Potentials*, vol. 20, no. 1, pp. 11-15, Feb-March 2001.

[3] C. Metz, "The Latest in Virtual Private Networks: Part I," *IEEE Internet Computing*, vol. 7, no. 1, pp. 87-91, Jan.-Feb. 2003.

[4] VPN Technologies: Definitions and Requirements. VPN Consortium. Available at: http://www.vpnc.org/vpn-technologies.html.

[5] B. Schneier, *Applied Cryptography, 2nd Ed.* New York, NY: John Wiley and Sons, 1996.

[6] C. Davis, *IPSec: Securing VPNs.* New York. NY: RSA Press, 2001.

[7] S. Kent and R. Atkinson: "Security Architecture for the Internet Protocol," RFC 2401, November 1998.

[8] S. Kent and R. Atkinson: "IP Authentication Header," RFC 2402, November 1998.

[9] S. Kent and R. Atkinson: "IP Encapsulating Security Payload (ESP)," RFC 2406, November 1998.

[10] "Data Encryption Standard," Federal Information Processing Standards Publication 46-2, December 30, 1993.

[11] "Advanced Encryption Standard," Federal Information Processing Standards Publication 197, November 26, 2001.

[12] X. Zhang, K. K. Parhi, "Implementation approaches for the Advanced Encryption Standard algorithm, " *IEEE Circuits and Systems Magazine*, vol. 2, no. 4, pp. 24-46, 2002.

[13] S. Morioka, A. Satoh, "A 10Gbps Full-AES Crypto Design with a Twisted-BDD S-Box Architecture," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 12, no.7, pp.686-691, July 2004.

[14] S. Frankel, R. Glenn, and S. Kelly, "The AES-CBC Cipher Algorithm and Its Use with IPsec," RFC 3602, September 2003.

[15] A. J. Elbirt, W. Yip, B. Chetwynd, and C. Paar, "An FPGA Implementation and Performance Evaluation of the AES Block Cipher Candidate Algorithm Finalists," *The Third AES Conference (AES3)*, New York. April 2000. Available at http://csrc.nist.gov/encryption/aes/round2/conf3/aes3papers.html.

[16] K. Gaj and P. Chodowiec, "Comparison of the hardware performance of the AES candidates using reconfigurable hardware," *The Third AES Conference (AES3)*, New York. April 2000. Available at http://csrc.nist.gov/encryption/aes/round2/conf3/aes3papers.html.

[17] M. Dworkin, SP 800-38A 2001, "Recommendation for Block Cipher Modes of Operation," Dec. 2001.

[18] M. Mcloone and J. V. McCanny, "Generic architectures and semiconductor intellectual property cores for advanced encryption standards cryptography," *IEE Proceedings – Computers and Digital Techniques*, Vol. 150, no. 4, pp. 239-244, July 18, 2003.

[19] R. Hobson and S. Wakelin, "An Area Efficient High Speed S-Box Method," in *Proceedings of the IEEE International Workshop of System on Chip*, 2005.

[20] M. Alam, W. Badawy, and G. Jullien, "A novel pipelined threads architecture for AES encryption algorithm," *Proceedings of the IEEE International Conference on Application-Specific Systems, Architectures and Processors*, July 17-19, 2002, pp. 296-302.

[21] Jhing-Fa Wang, Sun-Wei Chang, and Po-Chuan Lin, "A novel round function architecture for AES encryption/decryption utilizing look-up table," *Proceedings of the IEEE 37th Annual International Carnahan Conference on Security Technology*, Oct. 14-16, 2003, pp. 132-136.

[22] R. Sever, A. N. Ismailglu, Y. C. Tekmen, M. Askar, and B. Okcan, "A high speed FPGA implementation of the Rijndael algorithm," *Euromicro Symposium on Digital System Design*, Aug. 31-Sept. 3 2004, pp. 358-362.

[23] A. Hodjat and I. Verbauwhede, "A 21.54 Gbits/s fully pipelined AES processor on FPGA," *12th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, April 20-23 2004, pp 308-309.

[24] X. Zhang and K. K. Parhi, "An efficient 21.56 Gpbs AES implementation on FPGA," *Conference Record of the Thirty-Eighth Asilomar Conference on Signals, Systems, and Computers*, 2004, Volume 1, pp. 465-470.