A COMPACT CRYPTOGRAPHIC PROCESSOR FOR IPSEC APPLICATIONS

A THESIS SUBMITTED TO
THE GRADUATE SCHOOL OF APPLIED MATHEMATICS
OF
MIDDLE EAST TECHNICAL UNIVERSITY

BY

ELİF BİLGE KAVUN

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR
THE DEGREE OF MASTER OF SCIENCE
IN
CRYPTOGRAPHY

SEPTEMBER 2010

Approval of the thesis:

A COMPACT CRYPTOGRAPHIC PROCESSOR FOR IPSEC APPLICATIONS

submitted by **ELİF BİLGE KAVUN** in partial fulfillment of the requirements for the degree of **Master of Science in Department of Cryptography**, **Middle East Technical University** by,

Prof. Dr. Ersan Akyıldız _____
Director, Graduate School of **Applied Mathematics**

Prof. Dr. Ferruh Özbudak _____
Head of Department, **Cryptography**

Prof. Dr. Ersan Akyıldız _____
Supervisor, **Department of Mathematics, METU**

Dr. Tolga Yalçın _____
Co-Supervisor, **Affiliated Faculty, METU**

Examining Committee Members:

Assoc. Prof. Dr. Cüneyt Bazlamaçcı _____
Department of Electrical and Electronics Engineering, METU

Prof. Dr. Ersan Akyıldız _____
Department of Mathematics, METU

Dr. Tolga Yalçın _____
Affiliated Faculty, METU

Assist. Prof. Dr. Ali Aydın Selçuk _____
Department of Computer Engineering, Bilkent University

Hakan Solmaz _____
Aselsan A.Ş., Ankara

Date: _____

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

Name, Last name: ELİF BİLGE KAVUN

Signature :

ABSTRACT

A COMPACT CRYPTOGRAPHIC PROCESSOR FOR IPSEC APPLICATIONS

Kavun, Elif Bilge
M.Sc., Department of Cryptography
Supervisor: Prof. Dr. Ersan Akyıldız
Co-Supervisor: Dr. Tolga Yalçın

September 2010, 142 pages

A compact cryptographic processor with custom integrated cryptographic coprocessors is designed and implemented. The processor is mainly aimed for IPSec applications, which require intense processing power for cryptographic operations. In the present design, this processing power is achieved via the custom cryptographic coprocessors. These are an AES engine, a SHA-1 engine and a Montgomery modular multiplier, which are connected to the main processor core through a generic flexible interface. The processor core is fully compatible with Zylin Processor Unit (ZPU) instruction set, allowing the use of ZPU toolchain. A minimum set of required instructions is implemented in hardware, while the rest of the instructions are emulated in software. The functionality of the cryptographic processor and its suitability for IPSec applications are demonstrated through implementation of sample IPSec protocols in C-code, which is compiled into machine code and run on the processor. The resultant processor, together with the sample codes, presents a pilot platform for the demonstration of hardware/software co-design and performance evaluation of IPSec protocols and components.

Keywords: Cryptography, Processor, GCC, IPSec, Compact

ÖZ

IPSEC UYGULAMALARI İÇİN KÜÇÜK ALANLI KRİPTOGRAFİK İŞLEMCİ

Kavun, Elif Bilge
Yüksek Lisans, Kriptografi Bölümü
Tez Yöneticisi : Prof. Dr. Ersan Akyıldız
Ortak Tez Yöneticisi: Dr. Tolga Yalçın

Eylül 2010, 142 sayfa

Entegre edilmiş işleme-özgü şifreleme alt-işlemcileriyle birlikte çalışan, az alan kaplayan bir işlemci tasarlanmış ve gerçeklenmiştir. İşlemci ağırlıklı olarak şifreleme işlemlerinde yoğun işlemci gücü gerektiren IPSec uygulamaları için amaçlanmıştır. Sunulan tasarımda, bu işleme gücü özel şifreleme alt-işlemcileri yoluyla elde edilmektedir. Bunlar, ana işlemciye genel bir esnek arabirim aracılığı ile bağlanmış olan bir AES çekirdeği, bir SHA-1 çekirdeği ve bir Montgomery modüler çarpıcısıdır. Tasarlanan işlemci çekirdeği, Zylin İşlemci Birimi (ZPU) çevirici programları kullanımına izin verecek şekilde, ZPU komut seti ile tamamen uyumludur. Gerekli olan komutların en küçük kümesi donanımsal olarak gerçeklenmiş, geri kalan komutların ise yazılımsal olarak benzeri yapılmıştır. Şifreleme işlemcisinin işlevselliği ve IPSec uygulamaları için uygunluğu, örnek IPSec protokollerinin C-kodu olarak gerçeklenmesi ile gösterilmiştir. Bu kodlar, makine koduna çevirilip işlemci üzerinde çalıştırılmıştır. Ortaya çıkan işlemci, örnek kodlarla beraber, donanım/yazılım ortak tasarımı ile IPSec protokol ve bileşenlerinin performans değerlendirme gösterimi için bir deneme platformu sunmaktadır.

Anahtar Kelimeler: Kriptografi, İşlemci, GCC, IPSec, Küçük alanlı

To mom and dad...

ACKNOWLEDGMENTS

I would like to thank all those people who have helped in the preparation of this study.

I am so grateful to the Institute of Applied Mathematics, my supervisor Prof. Dr. Ersan Akyıldız and my co-supervisor Dr. Tolga Yalçın for giving me the M.Sc. opportunity. My special thanks go to Dr. Yalçın, for guiding me with patience and his support that provided me determination throughout this study.

I would like to thank Percello Ltd. for supporting this work as an industrial partner.

I would also like to thank my examining committee members Assist. Prof. Dr. Ali Aydın Selçuk, Assoc. Prof. Dr. Cüneyt Bazlamaçcı and Hakan Solmaz for their time they spared for me.

Very special thanks to all my dear friends for their support and patience during my studies.

And, I would like to thank my family who has always given me endless support, care and help in all my decisions. I owe them what I am today, so I would like to dedicate this study to my dear mom and dad.

Thank you...

TABLE OF CONTENTS

LIST OF TABLES

TABLES

LIST OF FIGURES

FIGURES

CHAPTER 1

INTRODUCTION

Today, computing technology is everywhere in our lives. It is hard to think of a world without computers, mobile phones, personal digital assistants (PDAs) and portable navigation devices. Due to this rise in the utilization of computing technology, the need of authentication and privacy has also increased. Governments**,** military and corporations collect a great deal of confidential information about employees, customers, their activities, and store this information on computers and transmit across various types of communication networks to other computers. The organizations and individual users need a way to keep this information confidential and even secure electronically. This is where cryptography comes into the picture. It is the tool to secure information.

Originating from the Greek words of kryptos (secret) and grapho (writing) [1], cryptography is the practice and study of hiding information [2]. From an engineer's point of view, it can be defined as the science of converting data into a scrambled code by means of a secure cipher, so that it can be stored or sent over a public or private media and decipher back to its original form whenever needed.

Cryptography has been in existence for around 3000 years [3]. Human beings have always needed to hide information for several reasons since the beginning of time. The earliest known "Ceasar" substitution ciphers [4,5] have evolved into today's modern cryptography and have become a critical tool in real-world applications. The growth tendency and popularity of cryptography have led to novel scientific research and engineering development. Several algorithms for countless applications have been implemented on both software and hardware platforms, and published in the scientific literature [6,7,8].

1.1 Motive

Hardware implementations of cryptographic algorithms can be performed via different approaches. One way is to implement an algorithm on application-specific custom hardware. Many works have been made on such hardware implementations [9,10]. This approach generally

results in an optimal hardware in terms of performance. However, it has its drawbacks, especially considering the fact that most of today's complex algorithms are implemented on application specific integrated circuits (ASICs) [11], which are costly in terms of design cycle and manufacturing effort. Manufacturing costs can be considerably reduced with the use of field programmable gate arrays (FPGAs) [12], but the long design cycles and the associated costs stay the same.

Another method is to use a microprocessor and write software code (generally, assembly code to get better results) implementing the target algorithm on it. This approach, while presenting a shorter turnaround, is far from being compact. For example, it takes several lines of code and even more clock cycles to complete a single finite field multiplication via software, while it can be executed in a single cycle using custom hardware [13]. There have been tremendous works to implement specific cryptographic processors to overcome this remedy [14, 15]. Such processors mostly offer cryptographic support through special instructions aimed to speed up basic functions or even implement complete cryptographic operations such as "AESENC" instruction present in the new AES enhanced Intel processors [16]. However, performance of this class of processors is limited to specific applications they are targeted for.

The idea presented in this thesis combines the advantages of these two approaches. Benefits of software run on microprocessor and custom hardware acceleration are unified in a hybrid fashion in order to provide a compact and fast solution. A compact microprocessor which implements Zylin Processor Unit (ZPU) [17] instruction set architecture (ISA) [18] is designed with a flexible plug-in interface through which several coprocessors capable of implementing various standard cryptographic algorithms are connected. The resultant microprocessor can implement cryptographic algorithms such as RSA [19], AES (Advanced Encryption Standard) [20] and SHA-1 (Secure Hash Algorithm) [21] via these dedicated coprocessors, which interact with it through a memory I/O operation based plug-in interface. Another advantage of this idea is that the plug-in interface is reconfigurable which allows addition of other coprocessors as well as removal of existing unused ones very easily. There is up to 25% reduction in the data processing capability (throughput) with respect to a hardware-only solution, which is quite acceptable given the flexibility and reconfigurability advantages of the architecture.

1.2 Previous Work

Many works have been made on this subject using either the custom hardware or software on microprocessor approach [22,23,24,25,26,27]. Fully custom implementations given in [22,23] present fast and compact designs, but unfortunately come with very long design cycle times and limited or even no reconfigurability at all. On the other side, the cryptographic processors make use of different instruction set architectures, and most of them specialize on a single algorithm or

application [24,25]. Therefore, neither approach is the best choice to obtain the right balance between a compact, fast and yet flexible design.

In [26] and [27], advantages of custom hardware and software on a single platform are combined by different approaches. Both methods demonstrate certain advantages over the previous hardware-only and software-only solutions. However, they too fail to provide a generalized and flexible solution as targeted by our design.

1.3 Target

In theory, a complex instruction set (CISC) [28] can be a good way to implement a processor. A complex instruction set is a computer instruction set architecture in which each instruction can execute several low-level operations such as a load from memory, an arithmetic operation and a memory store, all in a single instruction. This can be done by implementing the whole set of the CISC architecture and it also has the advantage of running different platforms on the processor. However, in practice, implementing the whole complex instruction set architecture is costly and not suitable for mobile applications such as mobile phones, PDAs, etc. [29]. It is also possible to design a compact, fast and low-power microprocessor which is suitable for mobile devices, using a reduced instruction set (RISC) [30] architecture much more efficiently. However, even "reduced instruction set" means an average of 16-32 bits wide instructions, resulting in a large program memory requirement for embedded applications. On the other hand it is possible to combine the advantages of both approaches in hybrid stack based ISA [31]. Such a solution is even a better choice with a minimal instruction set which is sufficient to run the flow control based tasks. In addition, a stack based architecture does not require any additional registers, reducing both the overall gate count and the program memory requirements.

Today's information technology (IT) applications demand high security, and make the cryptographic support an essential component for IT devices. The major goal of this study is to develop a compact, fast and low-power cryptographic microprocessor especially targeting mobile applications, which demand security in a constrained and low-cost environment. The cryptographic support can be provided by means of cryptographic instructions, which is not always desirable due to the practical impossibility of implementing a complete ISA that can implement several algorithms with a degree of performance close to custom hardware. A reconfigurable approach is better for mobile needs, as it allows implementation and integration of various cryptographic coprocessors for dynamically varying system requirements. A flexible plug-in support built into the microprocessor gives the opportunity to choose and add any desired coprocessor. This implies that a microprocessor with a flexible plug-in interface be implemented together with a minimal set of cryptographic coprocessor in order to get a compact, fast, low-power and yet reconfigurable secure mobile microprocessor.

1.4 Approach

In order to achieve the main goal of this study, we investigate a coprocessor based microprocessor design which connects custom crypto accelerators (coprocessors) to the main processor via a memory I/O based plug-in interface. The main processor has a stack-based ISA based on the ZPU instruction set architecture (ISA). Choosing ZPU instruction set architecture also provides a wide tool support [17] for both software and hardware development and verification. The processor can only implement a minimal set of instructions, which are sufficient for the event flow control of all the supported cryptographic coprocessors. The software code for the processor can be manually written in native assembler or can be generated from C/C++ code via GNU C compiler (GCC) [32].

In the present version of the design, three different cryptographic coprocessors are defined. The three coprocessors implement the Montgomery modular multiplication (MMM) [33-36] for RSA public-key cryptography algorithm, AES (Advanced Encryption Standard) and SHA-1 (Secure Hash Algorithm) algorithms. In its current form, the microprocessor is capable of implementing the Internet Protocol security (IPSec) protocol suite [37].

The functionality of the microprocessor and its coprocessors is tested using carefully selected applications for each of the coprocessors: basic RSA algorithm implementation using the Montgomery modular multiplication (MMM) coprocessor, hash-based message authentication code (HMAC) [38] using the SHA coprocessor and CCM mode [39] authenticated encryption using the AES coprocessor. The selected algorithms are implemented as software, making called to the respective coprocessors.

1.5 Thesis Outline

After the brief introduction in this chapter, the IPSec protocol suite is summarized in Chapter 2.

ZPU architecture and instruction set are presented in Chapter 3. This chapter continues with the implementation details of the custom ZPU compatible processor, including the security considerations and software development tools.

In Chapter 4, AES (Advanced Encryption Standard), SHA (Secure Hash Algorithm) and RSA algorithms are summarized. Implementation of coprocessors for each these algorithms are also presented in detail in this chapter.

Integration of coprocessors to the main processor, and the plug-in interface is explained in Chapter 5.

Chapter 6 presents IPSec protocol suite examples implemented on the designed cryptographic processor. The hardware/software partitioning of the algorithms, which is briefly introduced in the interface discussion, is also explained in this chapter in detail by means of actual examples.

Finally, Chapter 7 summarizes the conclusions and future directions for the continuation of the research presented in this thesis, followed by the References.

CHAPTER 2

INTERNET PROTOCOL SECURITY (IPSec)

The Internet Protocol (IP) is the protocol that is used for data communication over the Internet. It is also referred to as the Transmission Control Protocol/Internet Protocol (TCP/IP) [40]. IP delivers distinguished protocol packets, which are usually referred to as datagrams, from the source host to the destination host based on their addresses, by means of addressing methods and structures for datagram encapsulation. The first version of addressing structure is referred to as Internet Protocol Version 4 (IPv4) [41], which is still the dominant protocol of the Internet. However, its successor, Internet Protocol Version 6 (IPv6) is nowadays being deployed actively worldwide [42].

The main disadvantage of IP is its lack of a general-purpose mechanism for ensuring the authenticity and privacy of data. IP datagrams are usually routed between devices over unknown networks; hence, any information in the datagrams can easily be intercepted and even changed. As a result of the inherent security weaknesses of IP and the increased utilization of Internet services for critical applications, IP Security (IPSec) protocols were developed [37].

At first, IPSec was developed for IPv6, but then it has been engineered to cover the security needs of both IPv4 and IPv6 networks. Its operation in both versions differs only in the datagram formats used for authentication header (AH) and encapsulating security payload (ESP). In our work, we focus only on IPv4.

2.1 IPSec Overview

The problem of the IP version 4 is the expected exhaustion of its address limits, which is due to the increase in the utilization of Internet beyond anyone’s expectations. When the first version of IP was developed, the internet was relatively private. Today it is truly public, which is causing more and more security problems. Several methods have been developed over years to cover security needs. The most effective solution was to allow security at the IP level so that all higher-layer protocols in TCP/IP could use it. The resultant technology, which brings secure communications to the IP, is called IP Security (IPSec) [43].

IPSec is a set of services and protocols that provide a complete security solution for an IP network. These services and protocols combine to provide various types of protection. Since IPSec works at the IP layer, it can provide protection for any higher-layer TCP/IP application or protocol without the need for additional security methods, which is a major strength. Among the protection services offered by the IPSec are:

- Encryption of user data for privacy,
- Authentication of the integrity of a message to ensure that it is not changed over networks,
- Protection against certain types of security attacks,
- Ability for devices to negotiate the security algorithms and keys required to meet their security needs,
- Different security modes to meet different network needs.

2.2 IPSec Operation and Core Protocols

When two devices want to communicate securely, they set up a secure path that may traverse across many insecure intermediate systems. To perform this engagement, these devices must satisfy certain rules:

- They must agree on a set of security protocols to use so that each one sends data in a format the other can understand.
- They must decide on a specific encryption algorithm to use in encoding data.
- They must exchange keys that are used to decode the data that has been cryptographically encoded.
- After background work is completed, each device must use the protocols, methods, and keys previously agreed upon to encode data and send it across the network.

In the realization of its operation, IPSec uses many different components and core protocols as shown in Figure 2.1. Because of this multi-technique and multi-protocol characteristic of IPSec, its main architecture and behavior of all the core components and protocols are not defined in a single Internet standard. Instead, a collection of continuously evolving Request for Comments (RFCs) [44] defines the architecture, services and specific protocols which are used in IPSec. Most important of these standards are listed in Table 2.1.

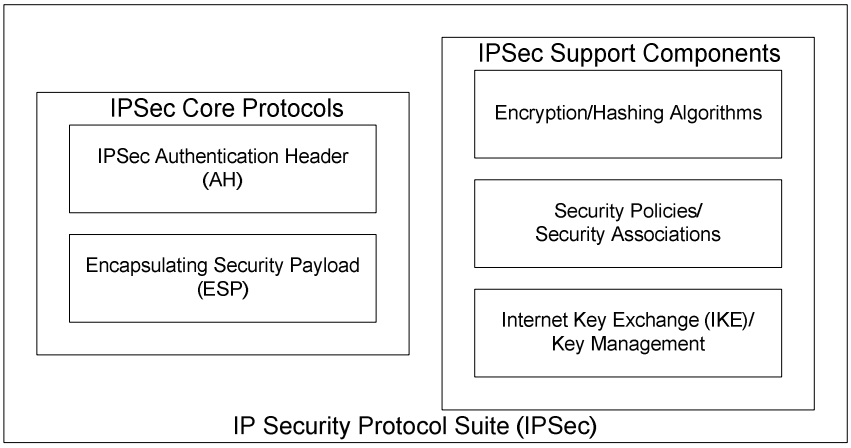

Figure 2.1 Overview of IPSec protocols and components.

Table 2.1 Important IPSec standards.

RFC Number	**Name**	**Description**
4301	Security Architecture for the Internet Protocol	The main IPSec document, describing the architecture and general operation of the technology, and showing how the different components fit together.
4302	IP Authentication Header	Defines the IPSec Authentication Header (AH) protocol, which is used for ensuring data integrity and origin verification.
4835	Cryptographic Algorithm Implementation Requirements for ESP and AH	Describes encryption and authentication algorithms for use by ESP and AH.
4303	IP Encapsulating Security Payload (ESP)	Describes the IPSec ESP protocol, which provides data encryption for confidentiality.
4306	The Internet Key Exchange (IKE)	Describes the IKE protocol that's used to negotiate security associations and exchange keys between devices for secure communications.

Two main pieces of IPSec, which actually manage information encoding to ensure security, are the authentication header (AH) and the encapsulating security payload (ESP) [43]. They are known as the core protocols of IPSec.

- IPSec Authentication Header (AH) provides authentication services for IPSec. It allows the recipient of a message to verify that the supposed originator of a message was actually the real one that sent it. It also allows the recipient to verify that intermediate devices over the network haven't changed any of the data in the datagram. AH also provides protection against replay attacks, where a message is captured by an unauthorized user and resent.
- Encapsulating Security Payload (ESP) provides privacy protection for the data. AH ensures the integrity of the data in datagram, but not its privacy. When the information in a datagram is private, it can be further protected using ESP, which encrypts the payload of the IP datagram.

2.3 IPSec Support Components

AH and ESP can not operate on their own. To function properly, these protocols need the support of several other protocols and services as can be seen in Figure 2.1. The most important of these services are:

- Encryption/Hashing Algorithms: AH and ESP do not specify an exact mechanism used for encryption, which makes them flexible to work with a variety of algorithms. Two common algorithms used with IPSec are the Secure Hash Algorithm 1 (SHA-1) for message hashing and the Advanced Encryption Standard (AES) for message encryption.
- Security Policies, Security Associations and Management Methods: IPSec is flexible in the decision of implementing the security which forces the devices to keep a record of the security relationships between themselves. This can be done using security policies and security associations of IPSec by providing ways to exchange security association information.
- Key Exchange Framework and Mechanism: Two devices which are exchanging encrypted information need to be able to share keys for decoding the encryption. Thus, they need a way to exchange security association information. A protocol called the Internet Key Exchange (IKE) provides these capabilities in IPSec.

2.4 IPSec Modes

Three basic implementation architectures can be used to provide IPSec facilities to TCP/IP networks [43]. These are:

- Integrated architecture,
- Bump in the stack (BITS) architecture,
- Bump in the wire (BITW) architecture.

The choice of implementation depends on the host device (end user or router) and impacts the specific way IPSec functions. There are two specific modes of operation that are related to these architectures: transport mode and tunnel mode [43].

IPSec modes are closely related to the function of the two core protocols, AH and ESP. Both protocols provide protection by adding a header containing security information to a datagram. The choice of mode determines which parts of the IP datagram are protected and how the headers are arranged to perform this operation. Modes describe how AH or ESP work and are used as a basis for defining other constructs, such as security associations (SAs).

2.4.1 Transport Mode

In transport mode, the protocol protects the message passed down to IP from the transport layer. The message is processed by AH and/or ESP and the appropriate header(s) are added in front of the transport (UDP or TCP) header. The IP header is then added in front of that by IP (Figure 2.2).

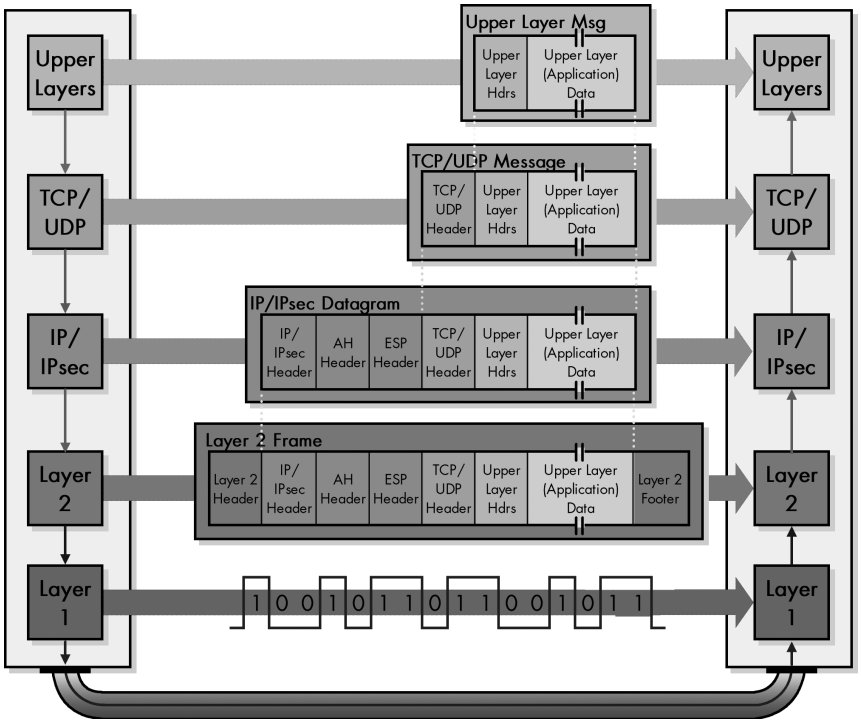

Figure 2.2 IPSec transport mode operation.

2.4.2 Tunnel Mode

In tunnel mode, IPSec is used to protect a completely encapsulated IP datagram after the IP header has already been applied to it. The IPSec headers appear in front of the original IP header and then a new IP header is added in front of the IPSec header. This means, the entire original IP datagram is secured and then encapsulated within another IP datagram (Figure 2.3).

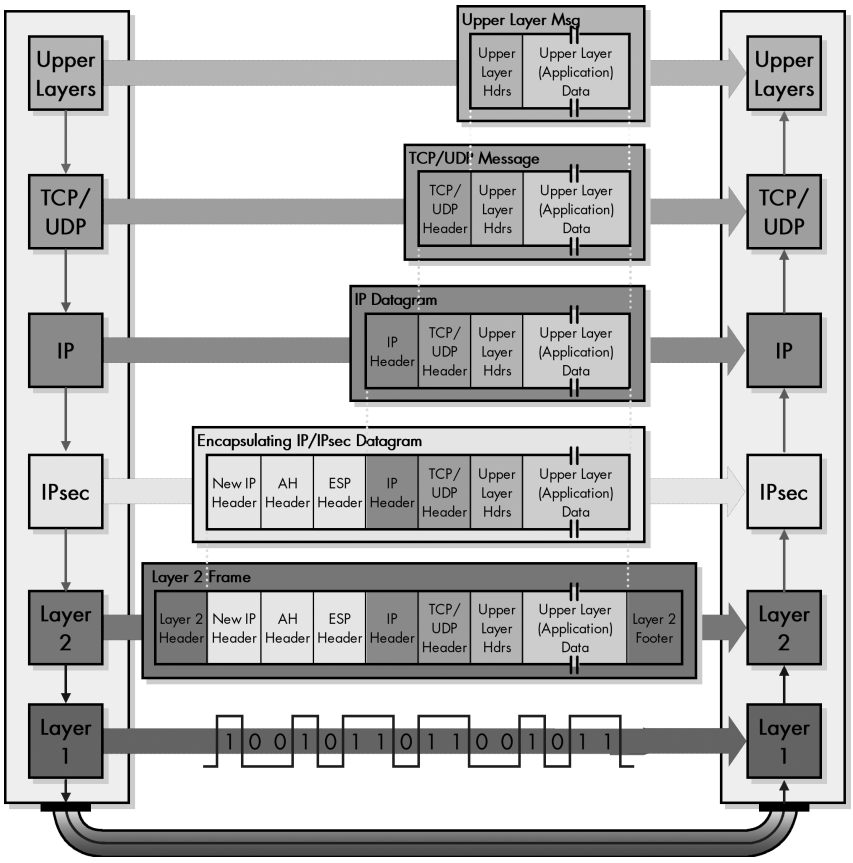

Figure 2.3 IPSec tunnel mode operation.

2.4.3 Comparison of Transport and Tunnel Modes

Tunnel mode protects the original IP datagram as well as its headers while transport mode does not take it as a whole. Hence, the order of the headers for two modes can be written as:

- **Transport Mode :** IP header, IPsec headers (AH and/or ESP), IP payload (including transport header)
- **Tunnel Mode :** New IP header, IPsec headers (AH and/or ESP), old IP header, IP payload

Using the three variables of mode (tunnel or transport), IP version (IPv4 or IPv6) and protocol (AH or ESP), eight basic IP packet combinations can be defined.

2.4.4 Relation of Modes with Architectures

Transport mode requires IPSec to be integrated into IP, because AH/ESP must be applied as the original IP packaging is performed on the transport layer message. This mode corresponds to the integrated architecture and is often the choice for implementations requiring end-to-end security with hosts that run IPSec directly.

Tunnel mode represents an encapsulation of IP within the combination of IP plus IPSec. So, it corresponds with the bump in the stack (BITS) and bump in the wire (BITW) implementations, where IPSec is applied after IP has processed higher-layer messages and has already added its header. This mode is a common choice for virtual private network (VPN) implementations, which are based on the tunneling of IP datagrams through an unsecured network such as the internet.

2.5 IPSec Authentication Header (AH)

Authentication header is one of the two core security protocols in IPSec. AH provides authentication of either all or part of the contents of a datagram through the addition of a header that is calculated based on the values in the datagram [43]. The parts of the datagram that are used for the calculation and the placement of the header, depend on the mode and IP version.

The operation of AH is simple, which is similar to the algorithms used to calculate checksums or perform cyclic redundancy checks (CRC) for error detection: The sender uses a standard algorithm to compute a checksum or CRC code based on the contents of a message. The computed result is transmitted along with the original data to the destination. There, the computation is repeated and the message is discarded if any discrepancy is found between the results.

The idea is same for AH, except that instead of a simple known algorithm, a special hashing algorithm is used together with a specific key known only to the source and the destination. Only the source and destination know how to perform the computation via a security association. On the source device, AH performs the computation and puts the result which is called the integrity check value (ICV) into a special header with other fields for transmission. The destination device does the same calculation using the shared key, and determines if any of the fields in the original datagram were modified.

The presence of the AH header verifies the integrity of the message, but it doesn't perform any encryption. Hence, ESP is used for providing privacy of the data. The calculation of AH is similar for both IPv4 and IPv6, except for the exact mechanism used for placing the header into the datagram and for linking the headers together.

In an IPv4 datagram, the Protocol field indicates the identity of the higher-layer protocol (typically TCP or UDP) which is carried in the datagram. This field points to the next header which is at the front of the IP payload. AH takes this value and puts it into its Next Header field, and then places the protocol value for AH itself (51 in dotted decimal) into the IP Protocol field. This makes the IP header point to the AH, which then points to whatever the IP datagram pointed to before.

In transport mode, the AH header is added after the main IP header of the original datagram. In tunnel mode, it is added after the new IP header that encapsulates the original datagram that's being tunneled. This is illustrated in Figure 2.4.

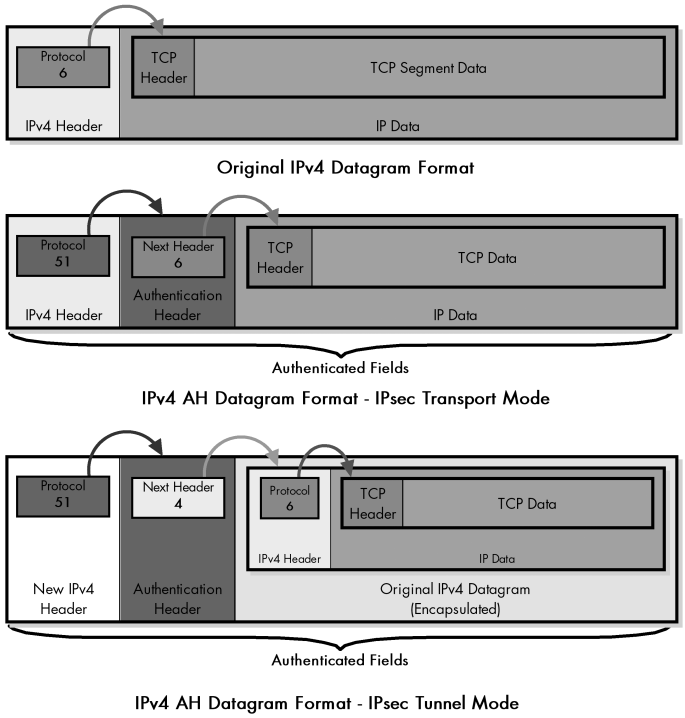

Figure 2.4 IPv4 datagram format with IPSec AH.

The format of AH is shown in Figure 2.5.

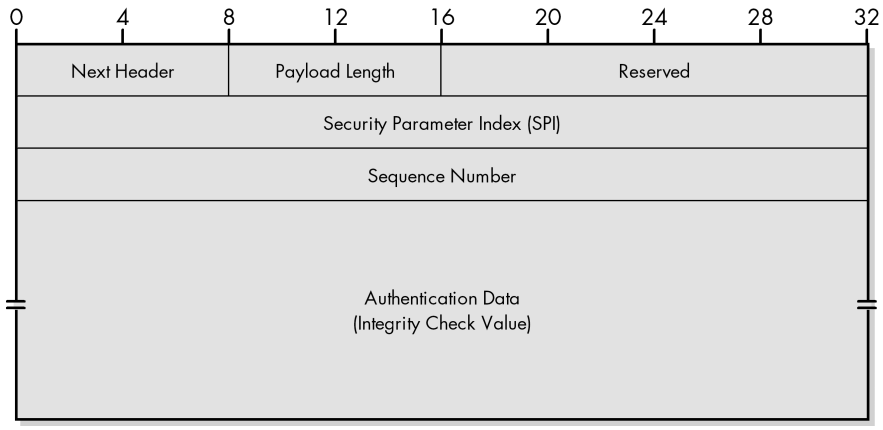

Figure 2.5 IPSec AH format.

The size of the Authentication Data field is variable to support different datagram lengths and hashing algorithms. Its total length must be a multiple of 32 bits. Also, the entire header must be a multiple of either 32 bits (for IPv4) or 64 bits (for IPv6), so additional padding may be added to the Authentication Data field if necessary.

2.6 IPSec Encapsulating Security Payload (ESP)

As mentioned before, data need not only be protected against possible changes over the network, but also against possible examination of its contents. For this reason, ESP protocol is used. The main job of ESP is to provide the privacy for IP datagrams by encrypting them. An encryption algorithm combines the data in the datagram with a key to transform it into an encrypted form. This is then repackaged using a special format and transmitted to the destination where it is decrypted using the same algorithm. ESP also supports its own authentication scheme like AH, or it can be used with AH.

ESP has several fields which are same as those used in AH, but they are packaged in a different way. Instead of a single header, ESP fields are divided into three components:

- **ESP Header:** Contains two fields, SPI and Sequence Number, and comes before the encrypted data. Its placement depends on if ESP is used in transport mode or tunnel mode.
- **ESP Trailer:** Placed after the encrypted data. It contains padding that is used to align the encrypted data through a Padding and Pad Length field. It also contains the Next Header field for ESP.

- **ESP Authentication Data:** Contains an ICV which is computed in a similar manner with AH protocol. This field is used when ESP's optional authentication feature is employed.

There are three basic steps performed by ESP: calculation of the header, the trailer and the authentication field.

- **Header Calculation:** As in AH format, the ESP Header field is placed after the normal IPv4 header. In transport mode, it appears after the IP header of the original datagram. In tunnel mode, it appears after the IP header of the new IP datagram which is encapsulating the original one. This is shown in Figure 2.6.

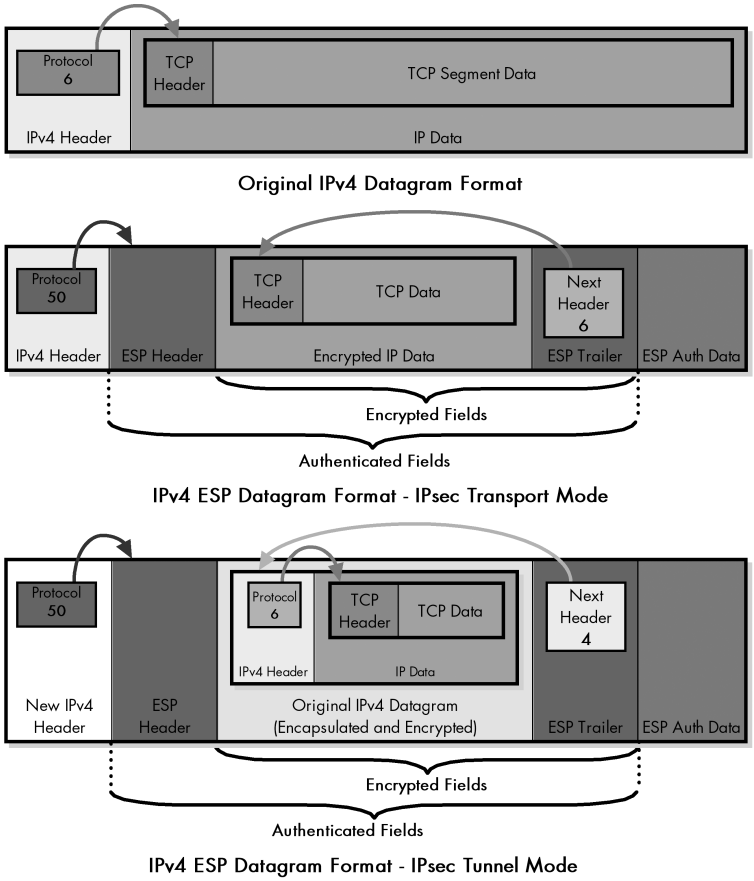

Figure 2.6 IPv4 datagram format with IPSec ESP.

- **Trailer Calculation:** The ESP Trailer field is appended to the data that will be encrypted, and then the payload (TCP/UDP message or encapsulated IP datagram) and the ESP trailer are both encrypted. However, the ESP header is not encrypted.
- **Authentication Field Calculation:** If the optional ESP authentication feature is being used, it is computed over the entire ESP datagram (except the Authentication Data field). This includes the ESP header, payload, and trailer.

The format of the ESP sections and fields is illustrated in Figure 2.7.

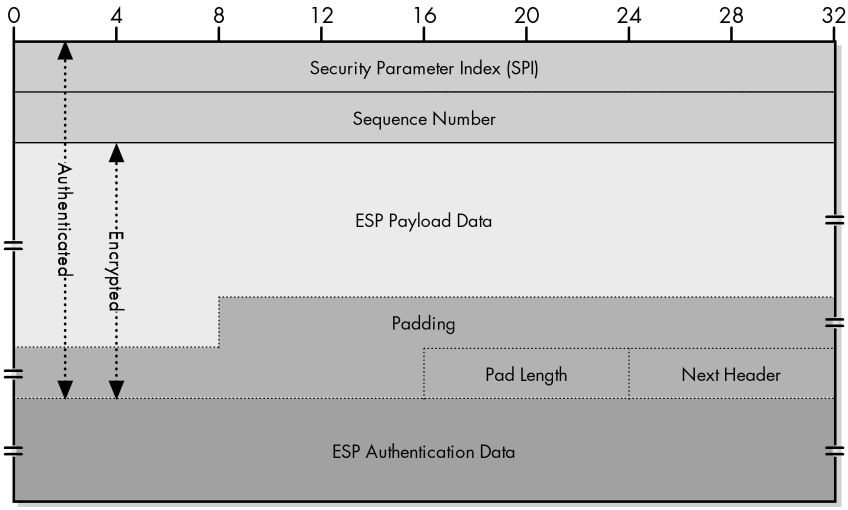

Figure 2.7 IPSec ESP format.

The Padding field is used when encryption algorithm requires, and/or to make sure that the ESP Trailer field ends on a 32-bit boundary, which means the size of the ESP Header field plus the Payload field, plus the ESP Trailer field must be a multiple of 32 bits. The ESP Authentication Data field must also be a multiple of 32 bits.

CHAPTER 3

SECURITY PROCESSOR DESIGN

In this thesis, the main target is the design and implementation of a compact processor core integrated with cryptographic coprocessors. The processor has to provide maximum support for the implementation of IPSec algorithms and be suitable for embedded applications in terms of area and power consumption. It also has to have GCC support, thereby providing C programming capability for the ease of use. In literature, there exist many different architectures and freeware processor designs. However, it is difficult to find an instruction set architecture (ISA) which is both compact, code-efficient and has GCC support. One architecture that satisfies all target features is the Zylin Processing Unit (ZPU) [17].

ZPU is a small, portable microprocessor core with GCC toolchain. It is an open source architecture, which allows deployments to implement any version of ZPU without running into license problems. The most important strength of the ZPU is that it is an extremely simple design, and therefore it is very easy to implement from scratch to suit specialized needs and optimizations [17]. Therefore, it is chosen as the target architecture for the processor core design of this thesis work.

However, the original ZPU code is not directly used. Instead, a new design is created from scratch which is one-to-one instruction set and code-compatible with the ZPU. The main difference in the design comes from the use of memories. The original ZPU core requires dual-port memories, with both read and write support in the same cycle on both ports. This is a very demanding requirement. Most FPGA architectures and ASIC technologies do not offer such memories. They either have dual-port memories with only read or write capability in the same cycle, or two-port memories with only-read capability on one port and only-write on the other port.

On the other hand, the extremely simple instruction set architecture of the ZPU can be easily implemented using only single-port memories. It may seem that such an implementation may result in longer execution times. However, as will be seen later in this chapter, this is not the case. Furthermore, the strength of the target cryptographic processor comes mainly from the

extremely fast execution of cryptographic algorithms via the integrated cryptographic coprocessors.

In the first section of this chapter, the ZPU instruction architecture and instruction set is presented. It is followed by sections presenting the details of the arithmetic logic unit and the instruction decoder, respectively. Then, the memory organization is given. Finally, security considerations and implementation details are summarized.

3.1 Architecture Overview and Instruction Set

ZPU is a stack-based processor. That means that it uses zero operand (unlike MIPS instruction set architecture [45], which has three operands). The elements which are at the top of the stack are used as operands. Using this approach, instructions can fit in 8 bits, which results in a very compact processor architecture.

The stack-based operation principle can be best explained by means of a simple instruction. Let's consider the AND instruction, which is defined as (mem[sp+1] = mem[sp+1] + mem[sp]; sp = sp + 1), in ZPU architecture. Basically; when an AND instruction comes, ZPU takes the topmost two values of the stack (pops) and ANDs them. Then, it adds the result of the operation to the top of the stack (pushes). As the stack is physically RAM-based, data is never taken out from the stack. Instead; the first data, which is pointed by the stack pointer (let sp=10. mem[10]), is taken and stored temporarily. Then, stack pointer is incremented by 1 and next data (mem[11]) is read. The AND operation is performed on the stored data and the present value of the memory addressed by the stack pointer. After the AND operation, the result is stored onto the top of the stack, which is pointed by the last value of the stack pointer (mem[11]). The input values are lost, and can not be used for next operations. However, it is possible to store those using different instructions and temporary registers, if necessary. At the end of the operation, the program counter is incremented by 1 and the next instruction is fetched. Then, the new instruction is decoded and necessary steps are performed according to the principles of stack operation.

The basic architecture that realizes the data flow for AND operation is shown in Figure 3.1.

The block diagram in Figure 3.1 only implements basic ALU instructions (like AND). It has to be modified to support other instructions (such as branch and stack memory load/store instructions). ZPU architecture is also flexible in terms of instructions. Different implementations of ZPU may support different numbers of instructions. The minimum required set (for proper execution of GCC compiled code) is composed of NOP, IM, LOADSP, STORESP, ADDSP, EMULATE, PUSHSP, POPPC, ADD, OR, AND, LOAD, NOT, FLIP, STORE and POPSP instructions.

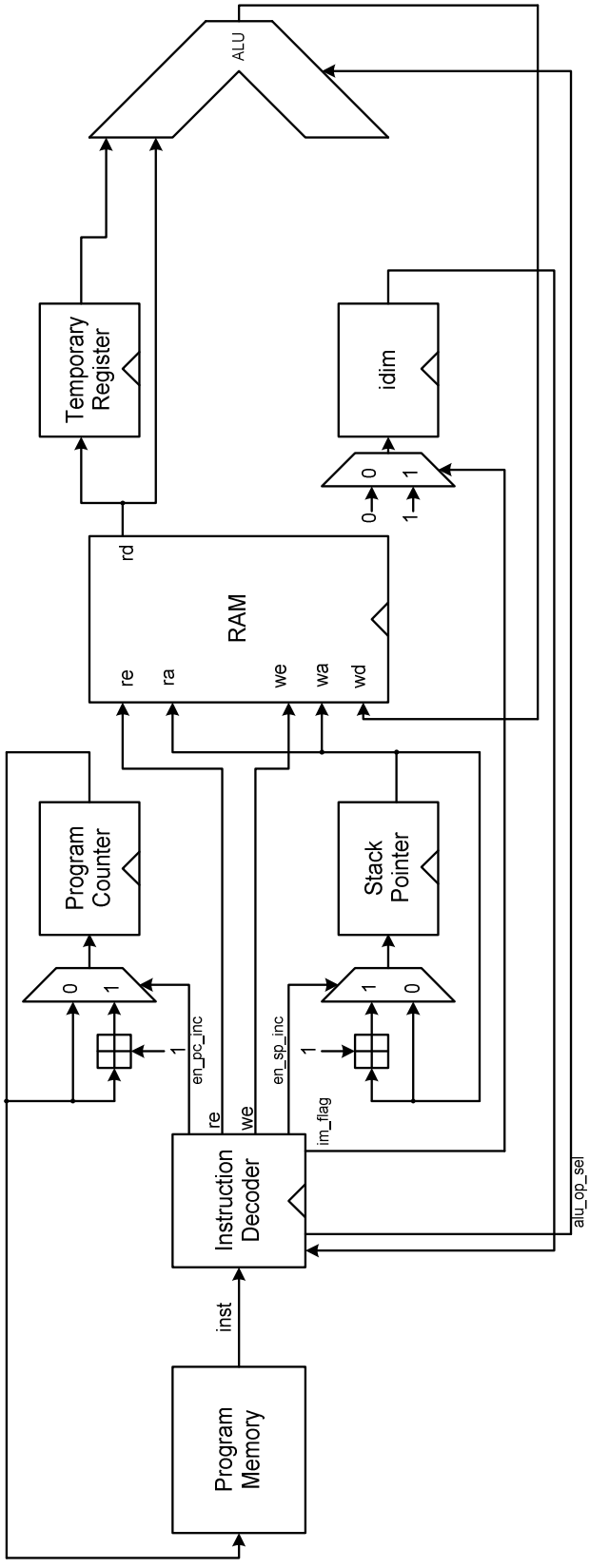

Figure 3.1 Basic architecture of ZPU for AND instruction.

Further instructions can be emulated in software using this minimum set. Naturally, they affect the efficiency of program execution. An emulated instruction requires execution of several hardware-coded instructions. Better efficiency can be reached by code profiling in order to determine the most commonly used instructions, and then implementing them as hardware-coded instructions, while keeping the emulated instruction count at minimum.

In this thesis, only the minimum required set of instructions is implemented in hardware. These instructions can be categorized in groups according to their functions. Dual operand ALU operations ADD, OR, AND can be categorized in one group, as ADD and OR work similarly to AND operation. Single operand ALU operations NOT and FLIP can be categorized in another group as they have the same working principle. However, the other instructions have their own structure, which makes them unsuitable to categorize. Table 3.1 shows a list of instructions and their functions.

Table 3.1 Instruction set.

MNEMONIC	OPCODE	HEX	OPERATION

IM X	1_xxxxxxx	-	if(~idim)
			{sp=sp-1; mem[sp]={{25{inst[6]}},inst[6:0]};
			idim=1}
			else
			{mem[sp]={mem[sp][24:0], inst[6:0]}; idim=1}

EMULATE X	001_xxxxx	-	sp=sp-1; mem[sp]=pc+1;
			pc=mem[@VECT_EMU+ inst[4:0]];
			fetch (used only by microcode)

STORESP X	010_xxxxx	-	mem[sp+ inst[4:0]*4] = mem[sp]; sp=sp+1
LOADSP X	011_xxxxx	-	mem[sp-1] = mem [sp+ inst[4:0]*4]; sp=sp-1
ADDSP X	0001_xxxx	(1x)	mem[sp] = mem[sp]+mem[sp+ inst[3:0]*4]
PUSHSP	0000_0010	(02)	mem[sp-1] = sp; sp = sp – 1
POPPC	0000_0100	(04)	pc=mem[sp]; sp = sp + 1
ADD	0000_0101	(05)	mem[sp+1] = mem[sp+1] + mem[sp]; sp = sp + 1
AND	0000_0110	(06)	mem[sp+1] = mem[sp+1] & mem[sp]; sp = sp + 1
OR	0000_0111	(07)	mem[sp+1] = mem[sp+1] | mem[sp]; sp = sp + 1
LOAD	0000_1000	(08)	mem[sp] = mem[mem[sp]]
NOT	0000_1001	(09)	mem[sp] = ~mem[sp]
FLIP	0000_1010	(0a)	mem[sp] = flip(mem[sp])
NOP	0000_1011	(0b)	no operation
STORE	0000_1100	(0c)	mem[mem[sp]] = mem[sp+1]; sp = sp + 2
POPSP	0000_1101	(0d)	sp = mem[sp]

In the table, sp is the stack pointer, pc is the program counter and mem[adr] is the RAM content addressed by adr. Note that, each instruction clears immediate data flag (idim), except IM. Also, each instruction updates program counter as pc++, except POPPC function.

The ZPU instructions can be completed in three cycles:

- **Fetch cycle:** To both decode the instruction and fetch the first operand from stack,
- **FetchNext cycle:** To store the first operand in the temporary register and fetch the second operand from stack,
- **Execute:** To execute the target operation and store the result back into the stack.

Some instructions can be completed in a single cycle, but the number of cycles is fixed to 3 cycles in order to simplify the overall design. This causes unnecessary cycles, however the resultant architecture is still more efficient that the original ZPU architecture, which may use up to 4 cycles per instruction. Furthermore, there is only a single memory access (either read or write) per cycle, allowing the use of single-port RAMs. The only penalty is separate program and data memories. However, this in practice, does not affect the overall resource utilization, especially in FPGAs. Another caused by single-port RAM use is the hardness (if not impossibility) of implementing pipelines. However, this is not a primary target for the current design.

Table 3.2 outlines the sequential and combinational operations performed in each cycle in order to realize each instruction.

Table 3.2 Instruction cycles.

	Fetch (001)	**FetchNext (010)**	**Execute (100)**
IM	pc <= pc + 1 If idim = 0: sp <= sp - 1 re = 0 If idim = 1: re = 1 ra = sp	we = 1 wa = sp If idim = 0: wd = inst[6:0] If idim = 1: wd = (rd<<7) || inst[6:0] idim <= 1	X
EMULATE	pc <= pc + 1 sp <= sp - 1 re = 1 ra = inst[4:0]<<5	we = 1 wa = sp wd = pc pc <= rd idim <= 0	X
STORESP	pc <= pc + 1 re = 1 ra = sp	sp <= sp + 1 we = 1 wa = sp + (4* inst[4:0]) wd = rd idim <= 0	X

	Fetch (001)	**FetchNext (010)**	**Execute (100)**
LOADSP	pc <= pc + 1 sp <= sp - 1 re = 1 ra = sp + (4* inst[4:0])	we = 1 wa = sp wd = rd idim <= 0	X
ADDSP	pc <= pc + 1 re = 1 ra = sp	tmp <= rd re = 1 ra = sp + (4* inst[3:0]) idim <= 0	we = 1 wa = sp wd = rd + tmp
PUSHSP	pc <= pc + 1 sp <= sp - 1 tmp <= sp	we = 1 wa = sp wd = tmp idim <= 0	X
POPPC	pc <= pc + 1 sp <= sp + 1 re = 1 ra = sp	pc <= rd idim <= 0	X
ADD, AND, OR	pc <= pc + 1 sp <= sp + 1 re = 1 ra = sp	tmp <= rd re = 1 ra = sp idim <= 0	we = 1 wa = sp wd = rd OP tmp
LOAD	pc <= pc + 1 re = 1 ra = sp	re = 1 ra = rd (= mem[sp]) idim <= 0	we = 1 wa = sp wd = rd
NOT, FLIP	pc <= pc + 1 re = 1 ra = sp	we = 1 wa = sp wd = OP (rd) idim <= 0	X
NOP	pc <= pc + 1	idim <= 0	X
STORE	pc <= pc + 1 sp <= sp + 1 re = 1 ra = sp	tmp <= rd (= mem[sp]) sp <= sp + 1 re = 1 ra = sp idim <= 0	we = 1 wa = tmp wd = rd
POPSP	pc <= pc + 1 re = 1 ra = sp	sp <= rd idim <= 0	X

This representation simply gives the timing diagrams of the instructions. Figure 3.2 is an example timing diagram, which shows the AND instruction explained before (also valid for ADD and OR instructions).

The hardware block of the overall ZPU core with all instructions can be easily constructed according to Table 3.2 and timing diagrams, as shown in Figure 3.3. The block diagram of the ZPU core consists of program memory, instruction decoder, program counter, stack pointer, RAM, immediate flag register “idim” and a temporary register. It is almost the same block as the

ALU operation block diagram in Figure 3.1, with additional circuitry for the realization of all target instructions.

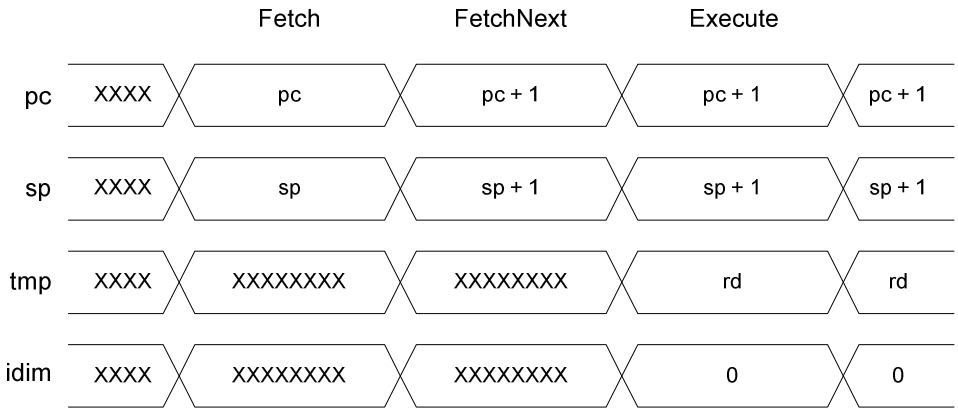

Figure 3.2 Timing diagram of AND instruction.

3.2 Arithmetic Logic Unit

As the name implies, arithmetic logic unit (ALU) of ZPU core performs the arithmetic and logic operations whenever needed. ALU unit is able to perform add, and, or, not, flip (reversing bits) and left shift operations. The operation select alu_op_sel comes from instruction decoder which identifies the instruction, and it selects the appropriate operation according to the instruction. Add unit is used for ADD and ADDSP instructions, and the other units are used for the corresponding instructions (same as their names), except shift units. First shift unit performs the instruction-specific 7-bit left shift operation of the IM instruction. The other one performs the instruction-specific 5-bit left shift operation of the EMULATE instruction. Figure 3.4 illustrates the arithmetic logic unit.

3.3 Instruction Decoder

Instruction decoder decodes the 8-bit instruction coming from the program memory according to the program counter and generates the control signals for individual processor blocks (ALU, RAM, stack pointer, program counter, etc.). Every instruction is executed in 3 clock cycles; thus, instruction decoder also guarantees that the next program memory value is not taken until the last cycle using the active signal and the busy signal (from coprocessors).

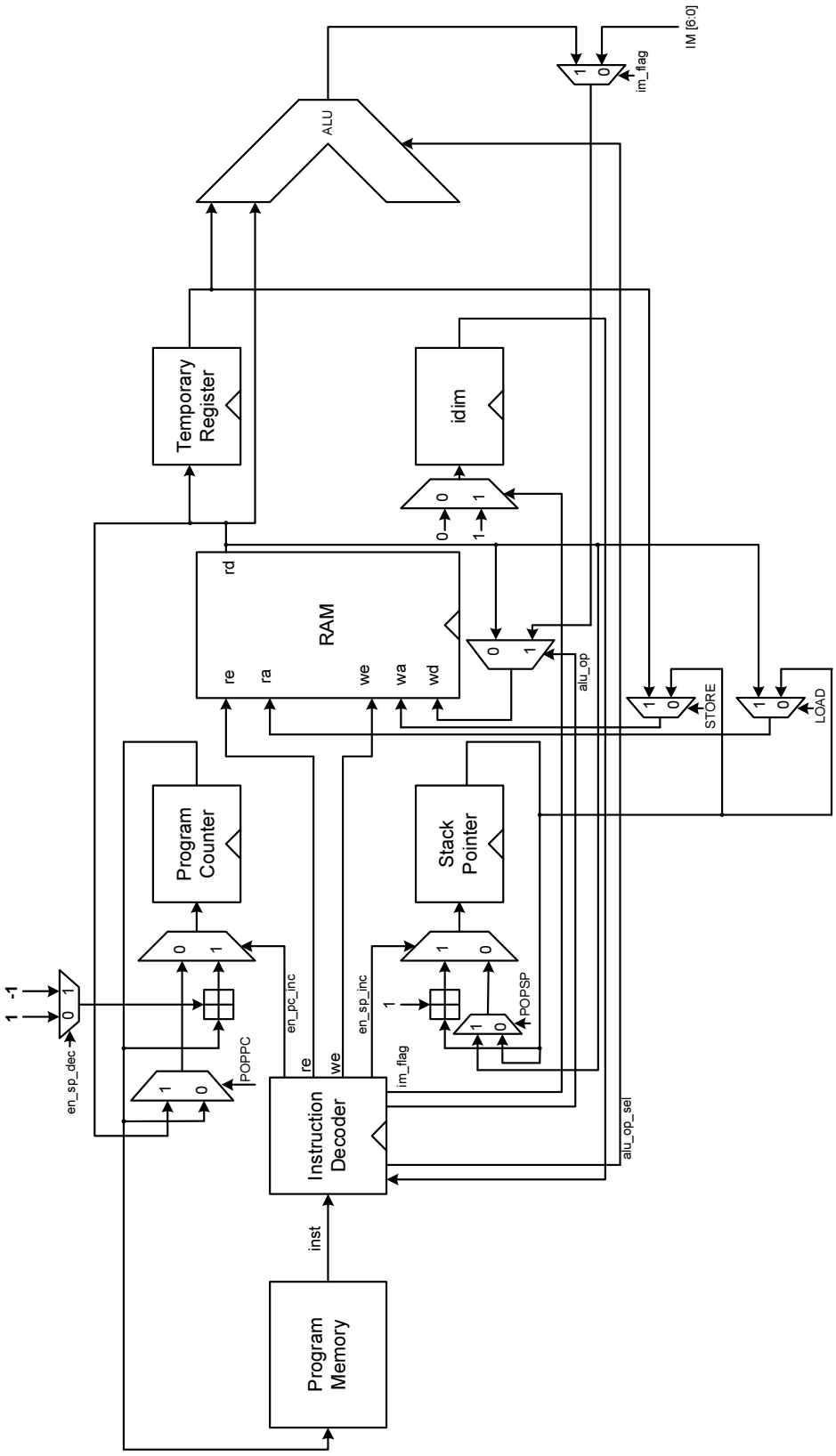

Figure 3.3 Block diagram of ZPU core.

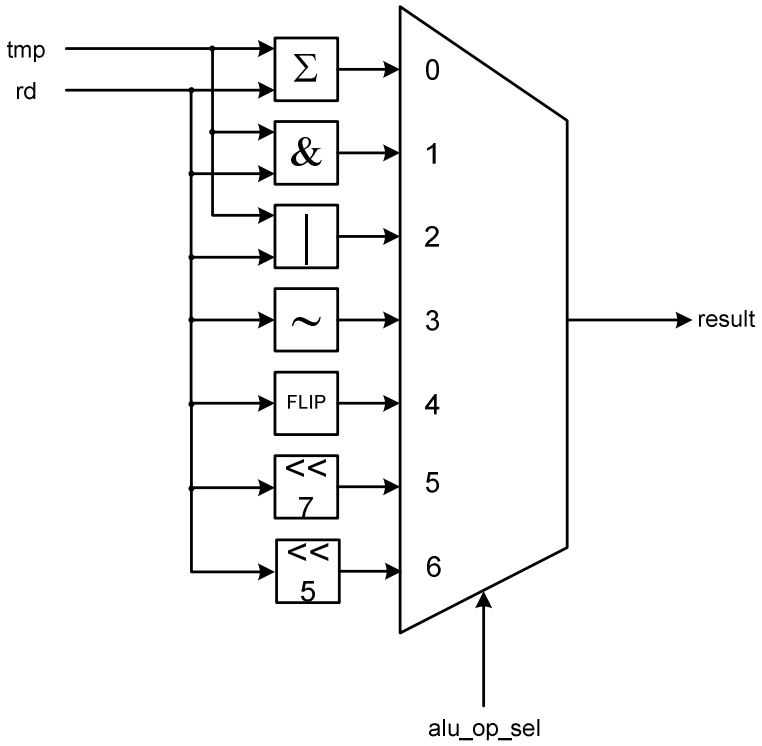

Figure 3.4 Arithmetic logic unit module.

The program counter is incremented by 1 at first cycle for all instructions, and except for emulate and poppc: Poppc takes the value of the read data from memory as the program counter value. In addition, stack pointer "increment-by-1" and "decrement-by-1" enables are defined for corresponding cycles and instructions as shown in Table 3.2. Read and write enables of the RAM are also defined according to this table and then sent to RAM.

The most significant bit of the 8-bit instruction is the "immediate" instruction (IM) select, if instruction[7] is 1, then immediate value will be taken. Otherwise, the most significant three bits will be controlled first. If instruction[7:5] is "001", "010" or "011"; then the instructions will be identified as "EMULATE", "STORESP" and "LOADSP", respectively. If they are all zero, then the next bit is controlled. In case that the fourth most significant bit is 1, the ADDSP operation is performed. In this instruction, an ALU operation enable is also sent to carry out the addition. In other cases, instructions are identified according to their least significant nibble, and the ALU enable is produced for ADD, AND, OR, NOT and FLIP. Note that, the ALU operation enable is also produced for the IM instruction, in case that the immediate value exceeds 7 bits and the left shift operation is required.

3.4 Memory Organization

The memory organization of ZPU, which covers the addresses of the coprocessors and the microprocessor, is quite simple. The least significant 16-bits of the 32-bit address space are used. Of these 16 bits, the most significant 4-bits (nibble) select the coprocessor and the remaining 3 nibbles address the specific location within the memory space of the selected coprocessor. Microprocessor is assumed to be the coprocessor-0. Therefore, addresses 0000-0FFF are used to address the microprocessor specific memory or memories.

In this implementation, AES, SHA-1 and MMM coprocessors are numbered as 1, 2 and 3, respectively, which corresponds to address spaces 1000-1FFF, 2000-2FFF and 3000-3FFF. The illustration of this memory organization is shown in Figure 3.5. This figure describes the real memory organization inside the RAM given in Figure 3.3.

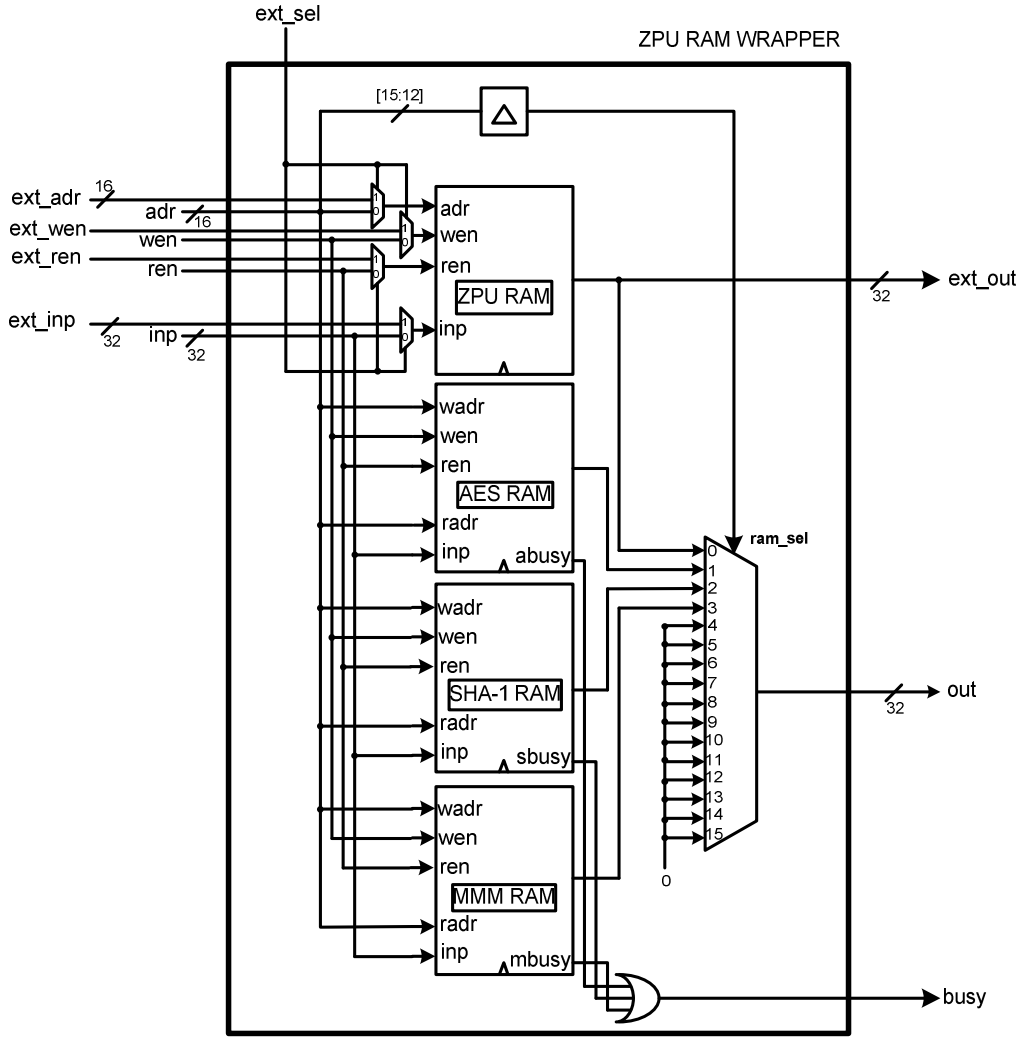

Figure 3.5 Memory organization.

Table 3.3 shows the address organization of each RAM. Here, each input space has 256 addresses. For example, the AES core input is addressed as 0x1A--, which means 2^8 = 256 bytes are available for an AES input (256x8 =2048-bits), despite the fact that the AES input is actually 128 bits (128/8=16 addresses) and using the addresses from 0x1A00 to 0x1A0F. In general case, this scheme provides convenience for addressing and simplifies the decoder/encoder logic.

Table 3.3 Address organization of RAM.

Address	Address Name	Description
0x00--	ZPU_rsv	Reserved places for ZPU
0x01--	ZPU_tmp	Temporary registers for processing
0x020-	CCM_M	CCM integrity check value size
0x021-	CCM _lm	CCM message length
0x022-	CCM _la	CCM additional authentication data length
0x023-	CCM _NNCs	Salt of CCM nonce
0x024-	CCM _NNCiv	Initialization vector of CCM nonce
0x025-	CCM _AAD	CCM additional authentication data
0x026-	CCM _Kd	CCM input key
0x027-	CCM _conf	CCM configuration register for key mode
0x030-	HMAC_lm	HMAC message length
0x031-	HMAC _Kd	HMAC input key
0x040-	RSA _e	RSA public key
0x041-	RSA_d	RSA private key
0x045-	RSA _N	RSA modulus
0x049-	RSA _K	RSA constant
0x04D-	RSA _len	RSA message length
0x04E-	RSA_conf	RSA configuration register for bit length
0x04F-	RSA_ED	RSA encryption/decryption select register
0x05--	MSG	Message
0x0E--	CCM _U	Authentication output of CCM
0x0F00	ZPU_RDY	ZPU core ready register
0x0FFF	ZPU_CSR	ZPU core command-status register
0x10--	AES_in	Input to AES core
0x11--	AES_out	Output of AES core
0x12--	AES_key	Key input to AES core
0x13--	AES_mod	Mode input to AES core
0x1F--	AES_CSR	AES core command-status register
0x20--	SHA_in	Input to SHA-1 core
0x21--	SHA_out	Output of SHA-1 core
0x22--	SHA_clr	Clear signal to SHA-1 core
0x2F--	SHA_CSR	SHA-1 core command-status register
0x30--	MMM_Ain	Input A to MMM core
0x31--	MMM_Bin	Input B to MMM core
0x32--	MMM_Cin	Input C to MMM core
0x33--	MMM_Yout	Output of MMM core
0x34--	MMM_mod	Mode input to MMM core
0x3F--	MMM_CSR	MMM core command-status register

As can be seen from the table, the most significant nibble is used to select the coprocessor and the following nibble is used to select a specific I/O location inside that coprocessor. Remaining 2 bytes are used to address the words of the selected I/O. This way, each I/O can be 256-bytes = 2048-bits long, which is also consistent with the maximum operand size of 2048-bits required for the MMM as the multiplicand, multiplier, modulus and the output is the multiplier result. With this addressing scheme, each coprocessor has 4096-byte address space, which is sufficient even for the most memory consuming RSA algorithm.

3.5 Security Considerations

In cryptographic hardware implementations, security leakage is a serious problem. Cryptanalysts have been developing many techniques to have successful attacks on these devices.

Side-channel cryptanalysis [46] is a kind of applied cryptanalysis, which uses the advantage of unintended physical leakage caused by a hardware implementation of a mathematically secure algorithm. Such a leakage can be sufficient to extract secret key material from cryptographic implementations. Another kind of implementation based attacks are fault analysis scenarios which aim to cause forced physical leakage.

Normally, mathematical cryptanalysis assumes that the cryptographic device only allows the use of input and output data of the cryptographic algorithm for cryptanalysis. However, other attacks are possible if the attacker has access to the device. These are called implementation attacks which target the cryptographic device. These attacks can be active attacks which range from changing the environmental conditions to the physical opening of the cryptographic device (probing and fault attacks), or passive attacks which observe the inherent physical leakage of the cryptographic device (side-channel attacks). The information leakage may be the power consumption of the device, electromagnetic radiation, timing information on the cryptographic service or obtained error messages.

In fault attacks, the changes towards extreme environmental conditions put the device under physical stress which may lead to a leakage. For example, malfunction can be caused by short-time pulses in the supply voltage or by freezing down the environmental temperature. Also, direct connections are made to an internal bus line to read out the cryptographic keys within the cryptographic device.

In side-channel attacks, the inherent physical leakage of the cryptographic device is used as an additional information channel for cryptanalysis. This physical leakage (for example power dissipation, timing information, etc.) can be captured externally and used to expose the secret key of the cryptographic algorithm by using standard statistical tools.

Generally, all cryptographic algorithms are assumed to be vulnerable to side channel cryptanalysis if there are not special precautions in the implementation [47]. The developer of a secure product has to defend the product against all possible attack paths. The efforts on these attacks are relatively low; however, the development of effective countermeasures is not a trivial task.

For side-channel attacks, countermeasures are easy to implement for timing analysis. It is generally sufficient to make sure that the execution time is data-independent. Power analysis attacks, which look at multiple specific intermediate values of the implementation, are harder to defeat. The countermeasure approaches can be hardware-based and/or software-based, algorithm-specific countermeasures.

Hardware countermeasures include special logic styles that minimize the data-dependent leakage. Also, noise generation and random process interrupts, which provide an internal timing de-synchronization, are used. Software countermeasures aim to avoid the occurrence of predictable intermediate results. Generally, internal randomization is used to mask the data representation used.

Countermeasures for fault analysis are relatively easy to side-channel attacks. It is required that the cryptographic device must check that the result obtained is correct. In the simplest way, this can be done by computing the same operation twice as it is appropriate for critical instruction paths, but an additional protection should be in place to detect modifications of security variables. Other countermeasures make use of certain control variables that are checked regularly. These countermeasures prevent from single faults even if they are precisely controlled, but it does not prevent from precisely controlled dual or multiple fault injections.

It should be noted that there are not any sufficient countermeasures in case the attacker has an ideal fault control using short-timed multiple fault injections. However, a combination of hardware and software countermeasures defeats a large number of existing attacks.

In the present implementation, resilience against the mentions attacks is not the primary concern. Instead, a compact design is sought. However, it is still possible to implement basic countermeasures at the expense of extra power consumption.

The first countermeasure is regarding the processor itself. The instruction cycles shown in Table 3.2 can be modified so that no registers are left idle in any cycle of any instruction. Every register can be assigned a dummy operation in a fashion that the overall operation flow is not affected. This scheme is illustrated in Table 3.4 in a few sample instructions with additional operations.

Table 3.4 Modified instruction cycles.

	Fetch (001)	**FetchNext (010)**	**Execute (100)**
IM	pc <= pc + 1 If idim = 0: sp <= sp - 1 re = **1** **tmp <= rand** If idim = 1: re = 1 ra = sp **sp <= rand** **tmp <= sp**	we = 1 **sp <= tmp** **tmp <= rand** If idim = 0: wa = sp wd = inst[6:0] If idim = 1: wa = **tmp** wd = (rd<<7) || inst[6:0] idim <= 1	**re = 1** **ra = rand** **sp <= tmp** **tmp <= ran**
STORESP	pc <= pc + 1 re = 1 ra = sp **sp <= rand** **tmp <= sp**	sp <= **tmp** + 1 **tmp <= rand** we = 1 wa= **tmp** + (4* inst[4:0]) wd = rd idim <= 0	X
LOADSP	pc <= pc + 1 sp <= **rand** re = 1 ra = sp+(4* inst[4:0]) **tmp <= sp - 1**	**sp <= tmp** **tmp <= rand** we = 1 wa = sp wd = rd idim <= 0	X

Bold characters in Table 3.4 represent modification and additions against power analysis attacks. With the additions to the IM instruction, it acts as a totally random instruction from a power analysis point of view. On the other hand STORESP and LOADSP instructions are now indifferentiable. The same registers are modified in the same cycles in both instructions.
However, the introduction of a random variable/function has to be noted. For the summarized scheme to work properly, a perfectly random number generated is required. Although hard on FPGA architectures, it is possible on ASICs with minimal effort. Most ASIC technologies offer true random number generators that make use of random noise of on-chip oscillators or amplifiers.

The second countermeasure scheme is continuously operating the coprocessors with random data. This can easily be achieved after simple modifications on the control circuitry. However, the true random number generator is again required. It should be made impossible for an outside observer to detect when the coprocessors are run with real or random data.

The approximate area increase in the instruction modification scheme is below 10 percent for the processor core, while there is area increase for the coprocessors in either case. However, running the coprocessor cores continuously increases the overall power consumption considerably. This may be undesired for most embedded applications.

3.6 Implementation Results

ZPU core is implemented on the smallest Virtex-5 device. Table 3.5 summarizes the results of the ZPU core without coprocessors. The slice count is 371 at a frequency of 101.6 MHz with a program memory of 8 kilobytes. If synthesized with a program memory of 2 kilobytes, the slice count is 105 at a frequency of 432 MHz. Our results are compared to the original small core implementation of ZPU with 8 kilobytes of program memory [17].

Table 3.5 ZPU core implementation results.

Xilinx Virtex-5 (xc5vlx30-3)	**Freq (MHz)**	**Area (slices)**	**Number of RAM Blocks**
With 8k program memory	101.6	371	5
With 2k program memory	432	105	0
ZPU small implementation [17]	202.8	170	8

3.7 Software Development Tools

ZPU processor has GCC toolchain support, including the GCC compiler, debugger and profiler, allowing the development of software in C. The compiler, debugger and profile are custom versions of the GCC tools re-compiled for ZPU architecture.

In the present work, the GCC tools are used to generate machine code that can be loaded directly into the processor's program memory. However, it is also possible to perform code profiling in order to determine the more and less frequently used instructions and modify the processor implementation by adding or removing hardware based instructions. In the present implementation only 16 instructions are hardware coded, the rest is all emulated instructions. The compiler is also capable of generating machine code for the emulated instructions and placing them into the program memory contents correctly. The microcodes for the emulated instructions are stored in a startup code file, which is passed to the compiler as an option. It is sufficient to edit this startup file in order to include or exclude certain instructions from the list of emulated instructions. This way, the program file generated will occupy less space for the emulation codes.

The ZPU toolchain produces highly compact code [17]. Below is a simple C code that creates two volatile arrays at fixed physical memory addresses and copies the contents or one array to the other word-by-word. This is in fact one of the most common operations performed in the implementation of IPSec protocol suites:

```
int main(void) {
  volatile int *a,*b;
  int i;
  a = (volatile int*)0x1000; /* a array at 0x1000 */
  b = (volatile int*)0x2000; /* b array at 0x2000 */
  for (i=0;i<100;i=i+4)
    b[i] = a[i] ;  /* Transfers contents of a to b */
}
```

When compiled with the GCC compiler with optimization options (`zpu-elf-gcc -O3 -S small.c`), this C code yield the following assembler code:

```
        .file   "small.c"
.text
        .globl  main
        .type   main, @function
main:
        im -2             // ZPU initialization
        pushspadd
        popsp
        im 0
        storesp 12
.L5:
        loadsp 8
        im 2
        ashiftleft
        im 8192           // b array address
        addsp 4
        im 4096           // a array address
        addsp 8
        loadsp 0          // Loop starts here
        load              // Load a[i]
        loadsp 8
        store             // Store b[i]
        im 4
        addsp 24          // Increment their addresses
        storesp 24
        storesp 8
        storesp 12
        storesp 4
        im 99             // Max value of i
        loadsp 12         // Load i
        lessthanorequal   // Check if i<=99
```

```
        impcrel .L5       // If yes return to loop
        neqbranch         // Return to start
        loadsp 0          // upon program completion
        im 0
        Store
        im 4
        pushspadd
        popsp
        poppc
        .size   main, .-main
        .ident  "GCC: (GNU) 3.4.2"
```

As seen in the above code, most instructions are already hardware coded instructions. The whole program is composed of only 25 instructions, which corresponds to 25 bytes of machine code plus the required emulation code (in this case 4 emulated instructions are used, resulting in 4×32=128 bytes of emulation code).

CHAPTER 4

CRYPTOGRAPHIC COPROCESSORS

This chapter outlines the algorithm and implementation details of the cryptographic coprocessors. The three coprocessors implement the AES encryption, SHA-1 hashing and Montgomery modular multiplication. Of these, AES encryption and SHA-1 hashing are must algorithms for IPSec, while Montgomery modular multiplication (MMM) is the computational component of the RSA algorithm used in Internet Key Exchange (IKE) protocol of IPSec protocol suite.

The flexible coprocessor interface explained before requires the coprocessors to behave as RAMs from the ZPU based main processor's point of view. Therefore AES and SHA-1 coprocessors are embedded into wrappers, which imitate single-port RAM behavior. This is not necessary for the MMM coprocessor, as it actually uses RAMs for operand and result storage.

The rest of this chapter is organized as follows:

In the first section, AES algorithm details are presented. This is followed by the introduction of a generic state machine model, which is then used to explain the implementation of the AES coprocessor.

In the next section, the same is done for the SHA-1: The algorithm explanation is followed by coprocessor implementation details, which are again presented by means of the generic state machine model.

The last section is devoted to the RSA algorithm and the MMM coprocessor. Both RSA encryption and decryption are summarized together with specific details of how the MMM coprocessor is used in the realization of them. Finally, the implementation details of the coprocessor block are outlined.

4.1 Advanced Encryption Standard (AES) Coprocessor

4.1.1 AES Algorithm

In cryptography, the Advanced Encryption Standard (AES) [20] is a block cipher which is adopted as an encryption standard by the US government. The cipher was developed by two Belgian cryptographers, Vincent Rijmen and Joan Daemen, and submitted to the AES selection process under the name "Rijndael". It was announced by National Institute of Standards and Technology (NIST) as U.S. FIPS PUB 197 on November 26, 2001 after a 5-year standardization process [48] and it became effective as a standard May 26, 2002. AES has been analyzed extensively and is now used widely worldwide as was the case with its predecessor, the Data Encryption Standard (DES) [49]. As of 2006, it is one of the most popular algorithms used in symmetric key cryptography. It is available by choice in many different encryption packages.

Unlike its predecessor DES, AES is a substitution-permutation network [50], not a Feistel network [51]. AES is fast in both software and hardware, is relatively easy to implement, and requires little memory. AES is currently being deployed on a large scale in hardware and software applications.

4.1.1.1 Description of the Cipher

AES algorithm has a fixed block size of 128 bits and a key size of 128, 192, or 256 bits. Most AES calculations are done in a special finite field. Due to the fixed block size of 128 bits, AES operates on a 4×4 array of bytes, termed as “state”. The 128-bit input to the cipher (0 to 127) is first grouped in 16-bytes (0 to 15), which are then put into 4×4 matrix (renamed as states 0,0 to 3,3). These states go through repetitions of processing steps that are applied to construct the rounds of keyed transformations between the input plain-text and the final output of cipher-text. The number of rounds, *Nr*, depends on the key size, which are 10, 12 and 14 for key sizes of 128, 192, and 256, respectively. Upon completion of all rounds, the resultant state matrix is renamed as output bytes (0 to 15), which form the 128-bit output vector. This scheme is illustrated in Figure 4.1.

The round function is parameterized using a key schedule. This key schedule consists of a one-dimensional array of four-byte words derived using a process known as the key expansion, described in Section 4.1.1.6.

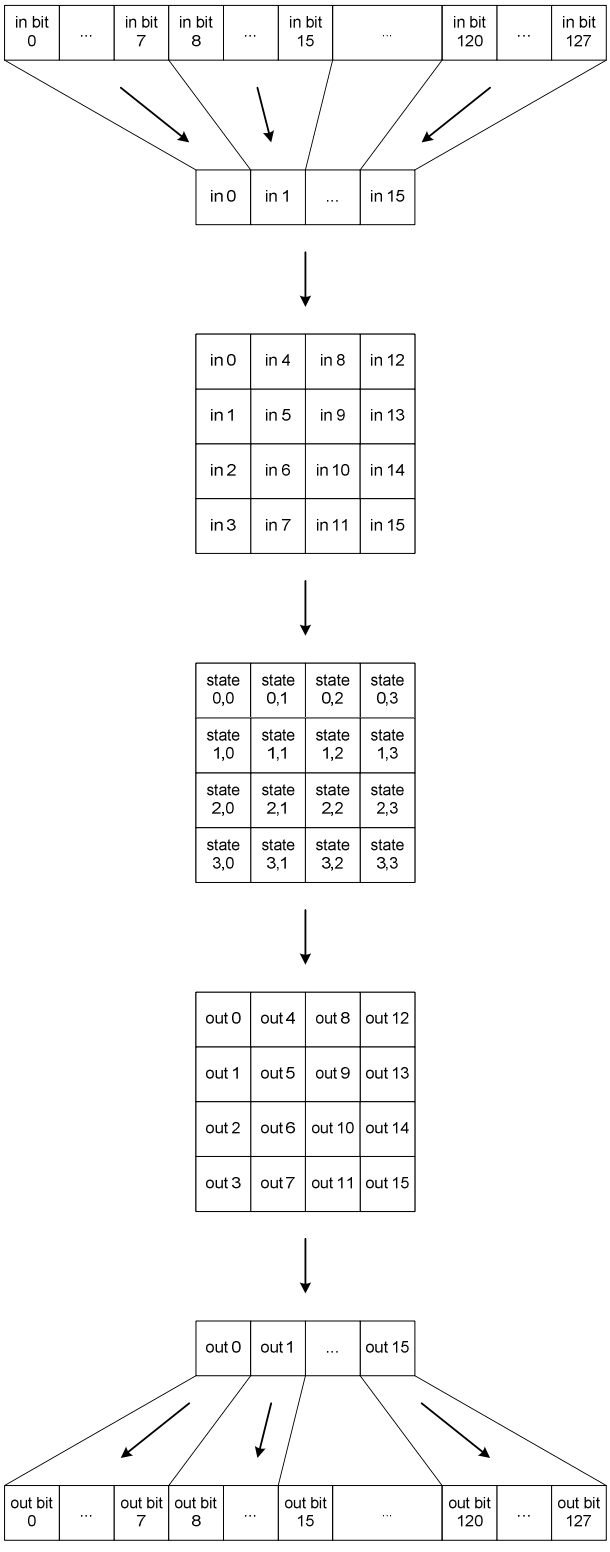

Figure 4.1 AES state illustration.

In the execution of the cipher algorithm, individual transformations – SubBytes(), ShiftRows(), MixColumns(), and AddRoundKey() – process the state bytes. These transformations are described in the following subsections.

The algorithm of the cipher can be expressed as follows:

```
Cipher(byte in[16], byte out[16], word w[4*(Nr+1)])

begin

    byte state[4,4]

    state = in

    AddRoundKey(state, w[0,3])

    for round = 1 step 1 to Nr–1
        SubBytes(state)
        ShiftRows(state)
        MixColumns(state)
        AddRoundKey(state, w[4*round,4*(round+1)-1])
    end for

    SubBytes(state)

    ShiftRows(state)

    AddRoundKey(state, w[4*Nr,4*(Nr+1)-1])

    out = state

end
```

In the pseudo code, the array *w[]* contains the key schedule.

4.1.1.2 The SubBytes Step

In the SubBytes step, each byte in the array is updated using an 8-bit substitution box, the Rijndael S-box. This operation provides the non-linearity in the cipher. The S-box used is derived from the following transformation which is also known as the affine transformation:

$$\begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \\ b_4 \\ b_5 \\ b_6 \\ b_7 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \end{bmatrix} \times \begin{bmatrix} d_0 \\ d_1 \\ d_2 \\ d_3 \\ d_4 \\ d_5 \\ d_6 \\ d_7 \end{bmatrix} + \begin{bmatrix} 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \\ 0 \end{bmatrix}, \quad (4.1)$$

where $[a_7a_6a_5a_4a_3a_2a_1a_0]$ and $[b_7b_6b_5b_4b_3b_2b_1b_0]$ are the input and output bytes, respectively, and $[d_7d_6d_5d_4d_3d_2d_1d_0]$ is the multiplicative inverse of the input byte. It should be noted that all the arithmetic operations are performed over GF(2^8) with an irreducible polynomial, p(x):

$$p(x) = x^8 + x^4 + x^3 + x + 1. \quad (4.2)$$

The multiplicative inverse over Galois field 2^8 (GF (2^8)) is known to have good non-linearity properties, which provides the nonlinearity property of the operation. To avoid attacks based on simple algebraic properties, the S-box is constructed by combining the inverse function with an invertible affine transformation. The S-box is also chosen to avoid any fixed points and also any opposite fixed points. The illustration of SubBytes operation is given in Figure 4.2.

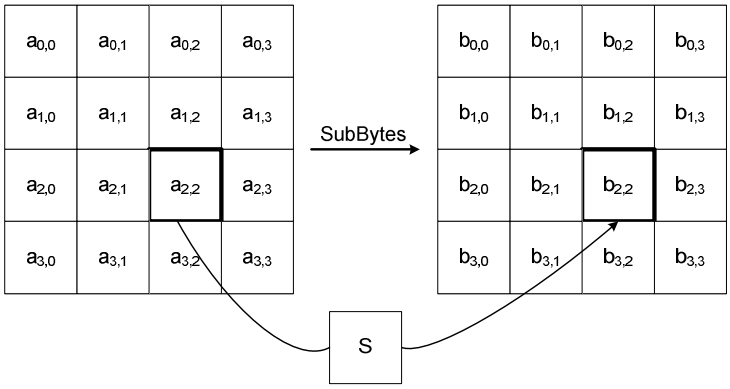

Figure 4.2 In the `SubBytes` step, each byte in the state is replaced with its entry in a fixed 8-bit lookup table S, as $b_{ij} = S(a_{ij})$.

4.1.1.3 The ShiftRows Step

The `ShiftRows` step operates on the rows of the state, it cyclically shifts the bytes in each row by a certain offset. The first row is left unchanged. Each byte of the second row is shifted one to the left. Similarly, the third and fourth rows are shifted by offsets of two and three, respectively. The illustration of ShiftRows operation is given in Figure 4.3.

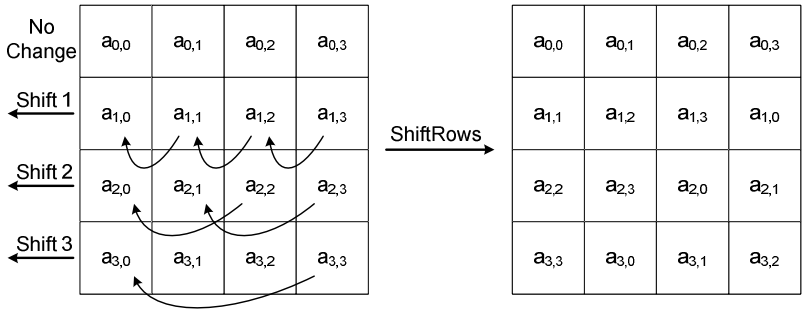

Figure 4.3 In the `ShiftRows` step, bytes in each row of the state are shifted cyclically to the left. The number of places each byte is shifted differs for each row.

4.1.1.4 The MixColumns Step

In the `MixColumns` step, the four bytes of each column of the state are combined using an invertible linear transformation. The `MixColumns` function takes four bytes as input and outputs four bytes, where each input byte affects all four output bytes. Together with `ShiftRows`, `MixColumns` operation provides diffusion in the cipher. Each column is treated as a polynomial over GF(2^8) and is then multiplied modulo $x^4 + 1$ with a fixed polynomial $c(x) = 3x^3 + x^2 + x + 2$ as shown in Equation 4.3.

$$\begin{bmatrix} b_{0,c} \\ b_{1,c} \\ b_{2,c} \\ b_{3,c} \end{bmatrix} = \begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix} \times \begin{bmatrix} a_{0,c} \\ a_{1,c} \\ a_{2,c} \\ a_{3,c} \end{bmatrix} \tag{4.3}$$

It should be noted that the elements of the transformation matrix are bytes represented as hexadecimal numbers. The illustration of MixColumns operation is given in Figure 4.4.

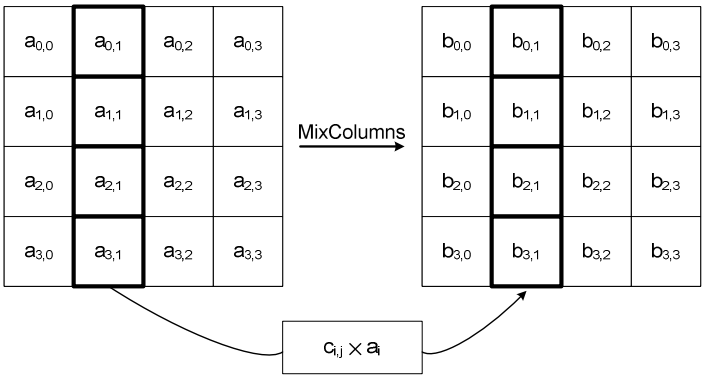

Figure 4.4 In the `MixColumns` step, each state column is multiplied with fixed polynomial c(x).

4.1.1.5 The AddRoundKey Step

In the `AddRoundKey` step, the subkey is combined with the state. For each round, a subkey is derived from the main key using AES key schedule and each subkey has the same size as the state. The subkey is added by combining each byte of the state with the corresponding byte of the subkey using bitwise XOR (which is the equivalent of addition over finite fields). The illustration of AddRoundKey operation is given in Figure 4.5.

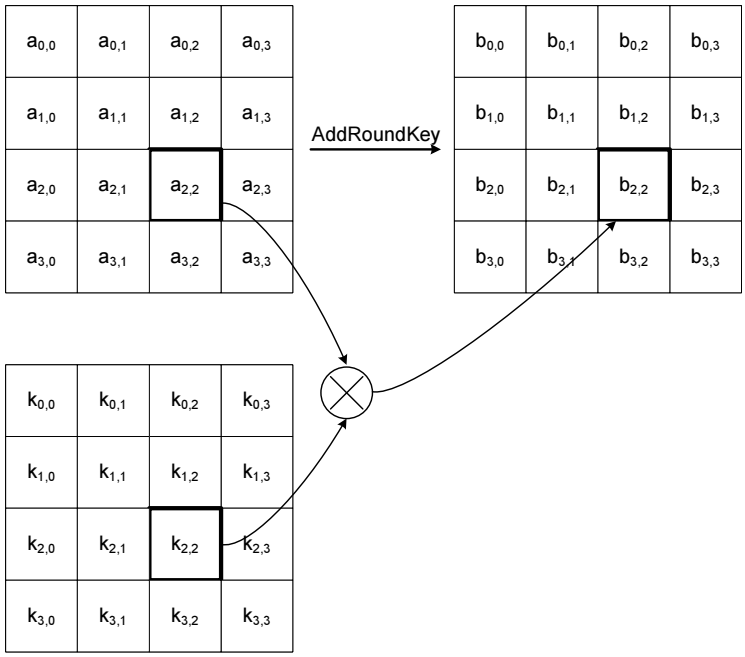

Figure 4.5 In the `AddRoundKey` step, each byte of the state is combined with a byte of the round subkey, using the XOR operation.

4.1.1.6 Key Expansion

The AES algorithm takes the cipher key, *K*, and performs a key expansion routine to generate a key schedule, namely the array *w[]* which is given in the cipher algorithm. The key expansion generates a total of $4\times(Nr+1)$ 32-bit words. The algorithm requires an initial set of 4 words, and each of the *Nr* rounds requires 4 words of key data. The resulting key schedule consists of a linear array of 4-byte words, denoted *w[i]*. The expansion of the input key into the key schedule proceeds according to the given pseudo code:

```
KeyExpansion(byte key[4*Nk], word w[4*(Nr+1)], Nk)

begin

    word temp

    i = 0

    while (i < Nk)
        w[i] = word(key[4*i], key[4*i+1], key[4*i+2], key[4*i+3])
        i = i+1
    end while

    i = Nk

    while (i < 4*(Nr+1)]
        temp = w[i-1]
        if (i mod Nk = 0)
            temp = SubWord(RotWord(temp)) xor Rcon[i/Nk]
        else if (Nk > 6 and i mod Nk = 4)
            temp = SubWord(temp)
        end if
        w[i] = w[i-Nk] xor temp
        i = i + 1
    end while

End
```

In the pseudo code, *Nk* is the key length which is 4, 6 and 8 words for input key lengths 128, 192 and 256 bits, respectively. SubWord is a function that takes a four-byte input word and applies the AES S-box to each of the four bytes to produce an output word. The function RotWord takes a word $[a_0, a_1, a_2, a_3]$ as input, performs a cyclic permutation, and returns the word $[a_1, a_2, a_3, a_0]$. The round constant word array, Rcon[i], contains the values given by $[x^{i-1}, \{00\}, \{00\}, \{00\}]$, with x^{i-1} being powers of x (x is denoted as $\{02\}$) in the field $GF(2^8)$ (note that i starts at 1).

4.1.2 Architecture Overview

We start the development of the architecture for the AES coprocessor by clearly identifying the processing steps and data flow through all rounds as shown in Figure 4.6. It is apparent from the figure that the *Nr* rounds in the definition of the algorithm maps to *Nr+1* round in the actual data flow, where the initial AddRoundKey step of the algorithm corresponds to round-0. Rounds 1 to *Nr*-1 are identical in terms of processing steps: SubBytes, ShiftRows, MixColumns and AddRoundKey. In the last round (round-*Nr*), MixColumns step is skipped.

The rounds can be directly mapped to a state machine, which first loads the input message and key into its registers as the initial state, and then processes the registered data via a combinational path, and updates the register data (state) at the end of each round. Output of the combinational path of the last round is loaded into the state register as the state machine output, which in this case is the AES encryption output.

For our state machines, we use a generic model, where data processed in three phases:

- **Initialization phase** is initiated by a "start" pulse, which puts the state machine into the active state. In the case of AES, this phase corresponds to round-0, where the input data (the message) is loaded into the state registers and the combinational path instantaneously generates the output of this round, which in fact is the result of the initial AddRoundKey step.
- **Iteration phase** is where rounds 1 to *Nr* are executed. The combinational output of each previous round is loaded into the state registers, and processed through the combinational path as the input of the next round. This phase takes *Nr* cycles to complete.
- **Finalization phase** is the state machine output generation and registering phase. Combinational result from round *Nr* is loaded into the state registers for the last time. Part or all the state register outputs becomes the state machine output. Combinational path output is not used. Instead, a single "ready" pulse output is generated to mark the completion of operation. The state machine quits the active state, and stays idle until the next "start" pulse.

The model summarized above is applied to the message data path and the key expansion data paths in parallel, resulting in a single united state machine. Round key output of the key expansion portion is taken directly from key state register outputs, in order to avoid additional combinational delay caused at the message side. The resulting state machine operation is illustrated in the pseudo code below, where sequential operations are denoted via "←" symbol, and combinational operations using simple equal signs.

```
1. Initialization (start comes)
    act    ← 1       :active status
    cnt    ← 0       :state counter
    Sreg   ← msg_inp :message register
    Kreg   ← key_inp :key register
    Rcon   ← 0x01    :Rcon (most-significant byte) register
    key    =  KeyRound[Kreg, Rcon]
    rc     =  Rcon
    state  =  Sreg ⊕ key

2. Iteration (cnt=1 to Nr)
    cnt    ← cnt + 1
    Sreg   ← state
    Kreg   ← key
    Rcon   ← rc
    key    =  KeyRound[Kreg, Rcon]
    rc     =  Rcon x 2
    state  =  MixColumns{ShiftRows[SubBytes(Sreg)]} ⊕ key

3. Finalization (cnt=Nr+1)
    act    ← 0
    cnt    ← Nr+1
    Sreg   ← state
    output =  Sreg

← : Sequential
=  : Combinational
```

As seen in the pseudo code, the state registers (both message, key and the Rcon register used in key expansion) are loaded with the input data (message and key) when “start” comes. This is in fact round-0 of the algorithm, as well as cycle-0 of the state machine. In addition, the active status flag (act) is activated, and the state counter (cnt) is cleared. Outputs from the combinational path are generated instantaneously as the round-1 inputs for both the message and key state registers. In the following cycles (1 to *Nr*), the state counter counts from 1 to *Nr*, the active flag stays active. Combinational outputs of each round are loaded to the state registers at the next round. Round-*Nr* (cnt=*Nr*) is the last cycle of operation (corresponds to the round in the algorithm, where MixColumns is skipped). Following this round is the finalization phase, where the state machine leaves the active state (act=0) and the combinational output from round-*Nr* is loaded into the state register for the last time, as the state machine (AES encryption) output. The key state and Rcon register values do not matter. How register and counter contents change is shown in the timing diagram in Figure 4.7.

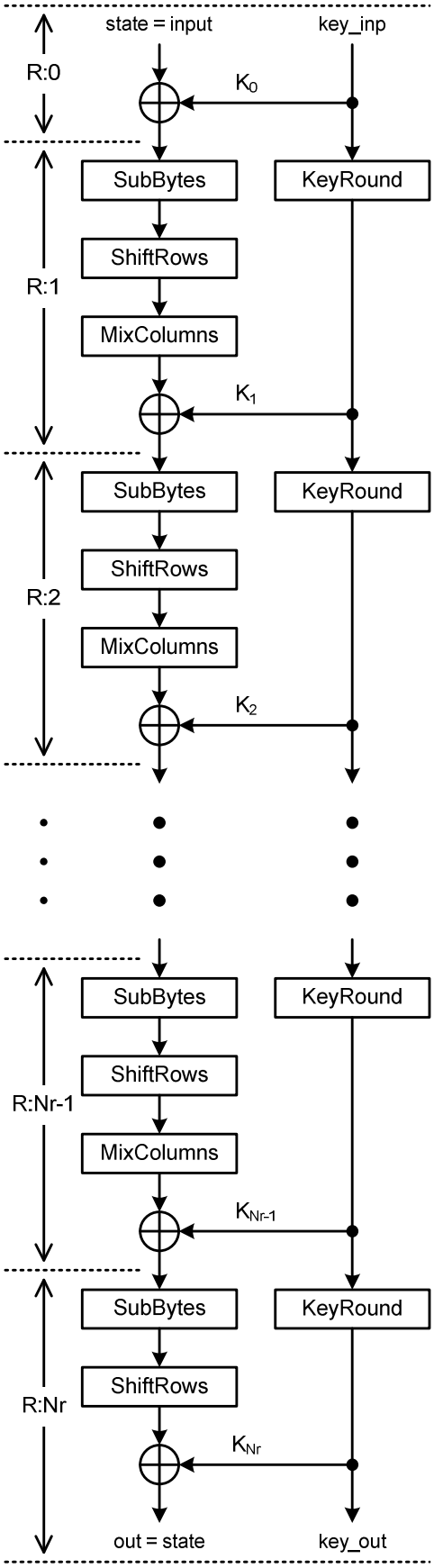

Figure 4.6 AES encryption algorithm data flow.

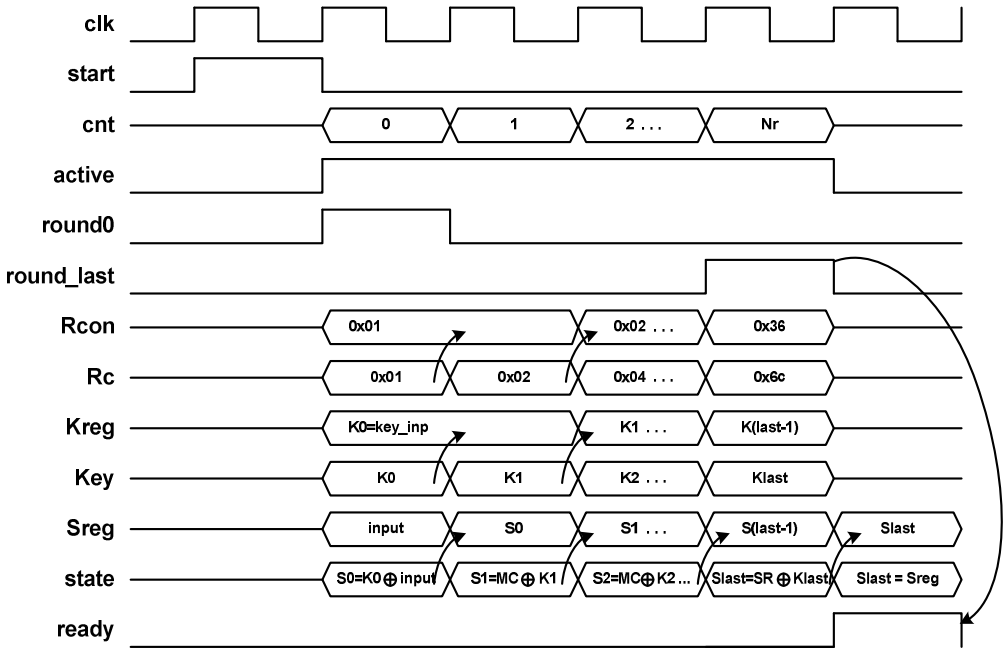

Figure 4.7 Timing diagram of AES block.

The presented pseudo-code together with the timing diagram above is mapped to the block diagram in Figure 4.8.

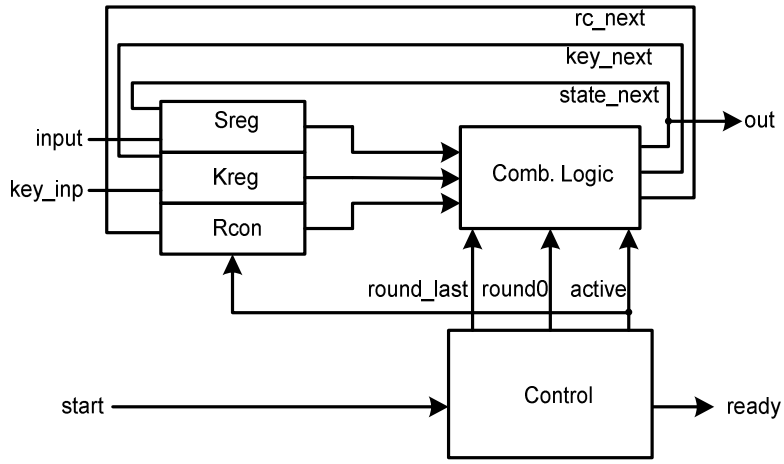

Figure 4.8 AES coprocessor block diagram.

Combinational logic required for key scheduling is quite complex and explained in detail in Section 4.1.3. Message processing data path is composed of *SubBytes*, *ShiftRows*, *MixColumns*,

and *AddRoundKey* steps multiplexed depending on the round being processed as shown in Figure 4.9.

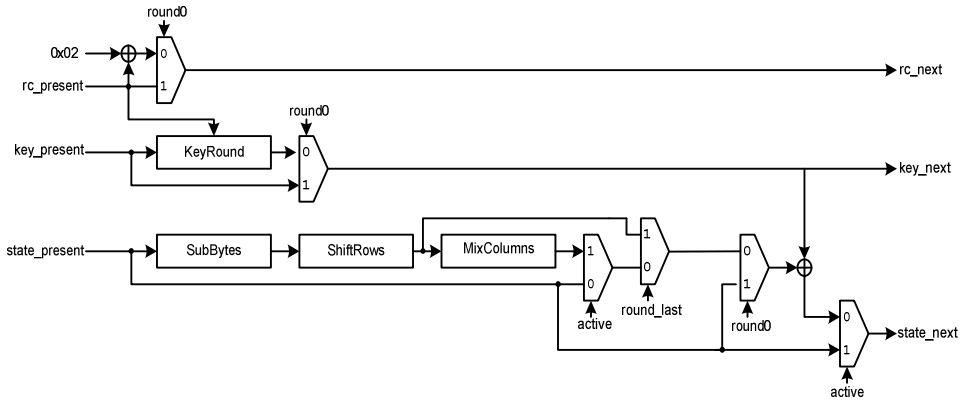

Figure 4.9 AES coprocessor combinational data path.

Another important issue about the AES coprocessor is the wrapper around it, which is not shown on the block diagram. In the design, a fully parallel implementation is favored in order to achieve the highest possible throughput. This requires the state and key registers to be 128 and 256-bits wide, respectively. However, AES coprocessor is used by the crypto processor with a 32-bits wide data bus. This means that data has to be written to and read from the AES coprocessor in 32-bit chunks. Therefore, a wrapper is implemented around the AES coprocessor.

128-bit state register is implemented as four 32-bit registers, mapped to specific addresses in the crypto coprocessor address space. When the address of one of the registers is selected, only the write enable for that specific register is enabled. In reading the result from the AES, a 4-to-1 multiplexer is used in order to select the specific part of the result.

The same logic also applies to the key register, except for a double buffer implemented at the input in order to preserve the original key, which is altered during the key scheduling process. In addition, no key register read addresses are implemented as it is deemed practically useless.

There are also two other registers required for the operation of the AES: mode register and command/status register (CSR). The mode register is a 2-bit write-only register used to store the chosen AES mode: 00 for AES-128, 01 for AES 192, and 10 for AES-256.

CSR is used to generate the “start” pulse. It is a zero-bit register. In other words, it is only an address in the crypto processor address space. When any write attempt is done to that address, the

AES “busy” register state goes high. From this register, a single “start” pulse is generated, which starts the AES operation. The “busy” register stays at high until it is cleared by the AES “ready” pulse. During this time, the crypto processor halts its regular program execution and waits for AES to complete its operation. When the busy register state goes low, the crypto processor continues with the next instruction, which is probably reading from the AES output addresses. This scheme together with other wrapper registers is shown in Figure 4.10.

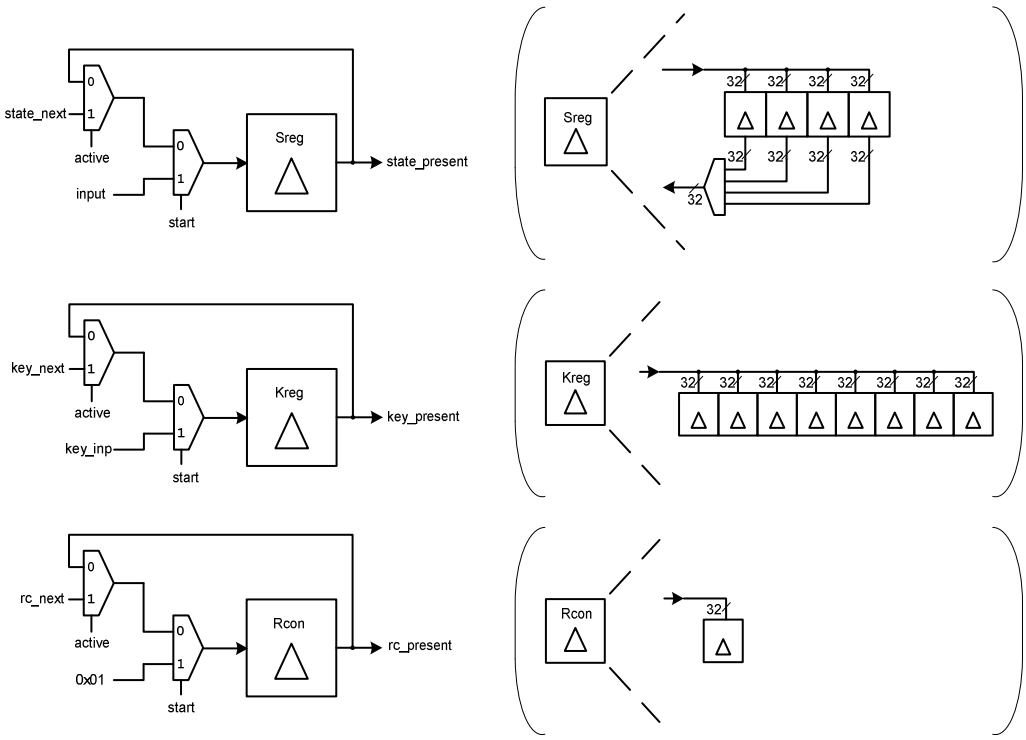

Figure 4.10 AES wrapper registers.

4.1.3 Key Scheduler

Key expansion module is implemented as a direct mapping of the key expansion pseudo-code. Depending on the key size, which is an input parameter, the module can generate key schedule for any key length of 128, 192 or 256-bit.

The key scheduler state machine pseudo-code was given as part of the AES engine. The important point is how the registers are used. As shown in Figure 4.11, the key input register is loaded in 32-bit words by the crypto processor. When the start pulse comes, all the 256-bits in the key input register are transferred to the key state register. When the active flag is high, the next

states for the registers are generated according to the round being processed, and loaded into them at the beginning of the next state.

Although both the input and state registers for key are 256-bits wide, they are partially used in case of AES-128 and AES-192 (only the leftmost 128 or 192 bits).

There is also the 8-bit Rcon register, which is initialized to $B_0 = 0x01$ with every "start" pulse, and then doubled at every step using multiplication over GF(2^8): $B_{i+1} = 2 \times B_i$. It is padded with 24-bit zeros before used in the actual key scheduling.

The *RotWord* and *SubWord* modules are cut-down versions of the *ShiftRows* and *SubBytes* in the data processing path. *RotWord* simply rotates the bytes in a 4-byte word by left, whereas the *SubWord* applied *SubByte* (s-box) function on all 4-bytes of a word in parallel.

4.1.4 SubBytes Module

SubBytes module is a composed of 16 parallel SubByte modules, each processing one byte of the input word. Therefore only the I/O interface of a single SubByte module is explained. The SubBytes and SubByte modules' schematics are given in Figure 4.12.

The easiest method to implement a SubByte module is to use a ROM based lookup table. However, in the current design, the SubByte module is implemented at the gate level via direct mapping of the arithmetic formula. For the target technology, gate level implementation is estimated to occupy less area. On the other hand, it is slower than a direct ROM implementation. But still, the total delay estimate is within the target limits.

The critical block for the SubByte module is the GF(2^8) multiplicative inverse block (inverter in short), which is explained in detail in [20]. The affine transform and inverter blocks are explained in the following subsections.

4.1.4.1 Affine Transform Module

The module operate on byte, transforming the input byte to the output byte via a matrix multiplication, where bit-level addition and multiplication correspond to logic level XOR and AND operations, respectively. The illustration of the block is shown in Figure 4.13.

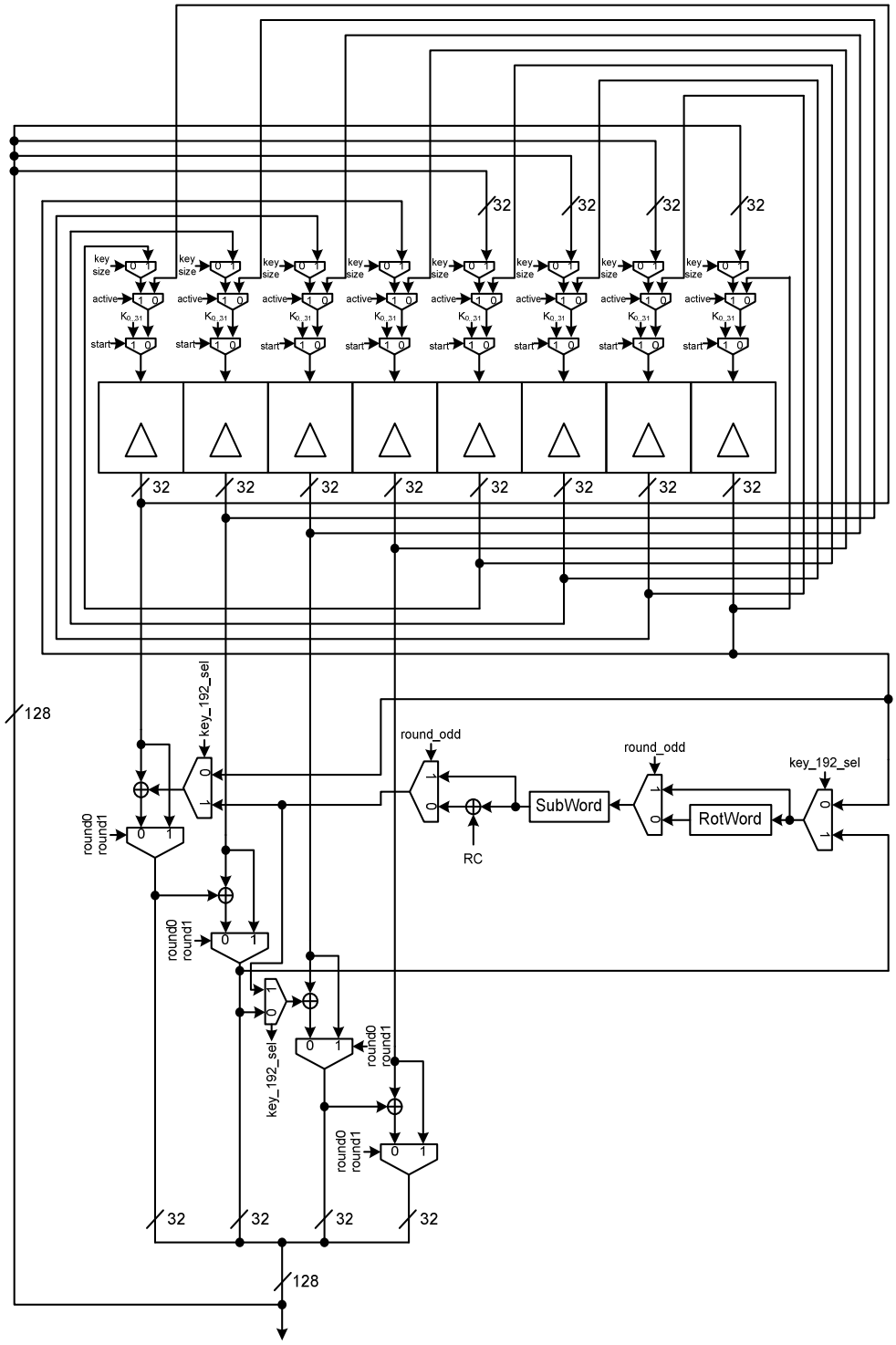

Figure 4.11 Key scheduler.

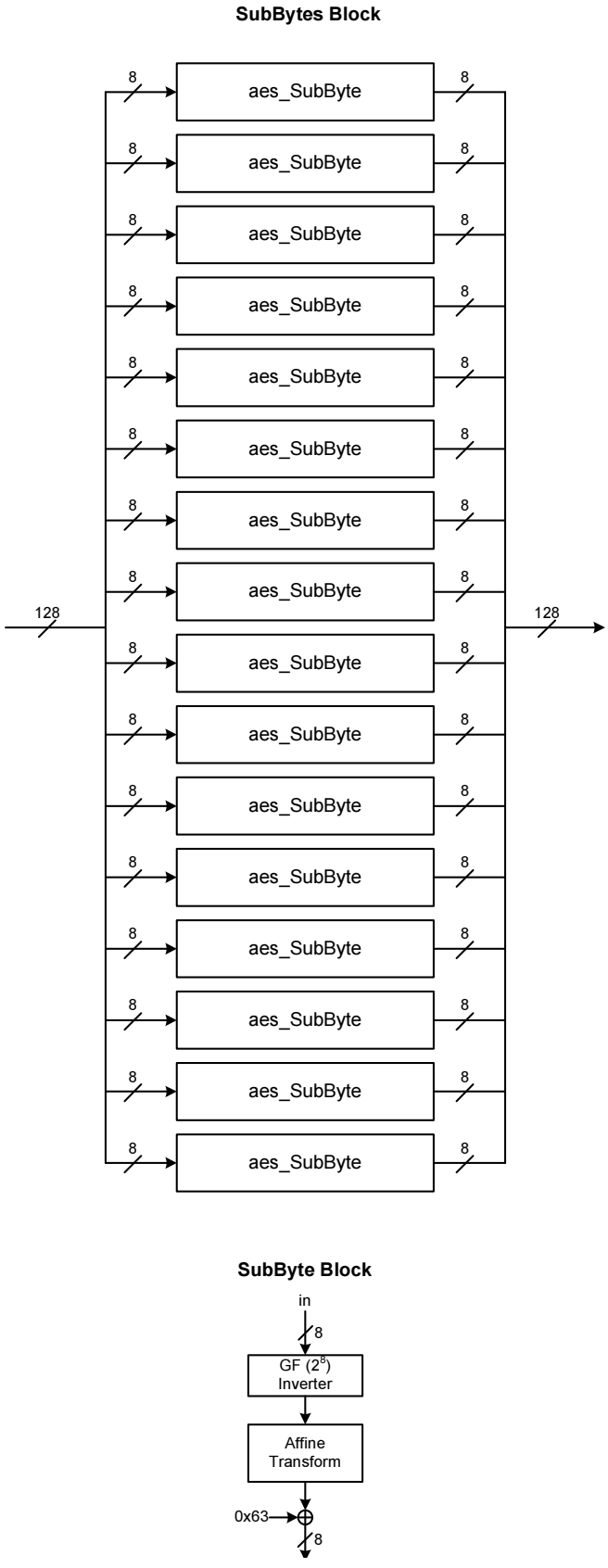

Figure 4.12 SubBytes and SubByte modules.

$$\begin{bmatrix} y_7 \\ y_6 \\ y_5 \\ y_4 \\ y_3 \\ y_2 \\ y_1 \\ y_0 \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \\ 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 \end{bmatrix} \times a \begin{bmatrix} 7 \\ 6 \\ 5 \\ 4 \\ 3 \\ 2 \\ 1 \\ 0 \end{bmatrix}$$

Figure 4.13 Affine transformation module.

4.1.4.2 GF(2^8) Multiplicative Inverse Module (Inverter)

Inverter module is implemented as a composite inverter [52]. In the composite inverter, GF(2^8) is processed as two separate finite fields: GF(2^4) and GF(4^2). In hardware, this is implemented by separating the byte into upper and lower nibbles, where each nibble represents a polynomial over GF(2^4). The two nibbles form a 2nd degree polynomial over the GF(4^2). This mode of operation reduces all the finite field operations to 4-bit level, simplifying the hardware implementation of arithmetic modules enormously.

In Figure 4.14, all the arithmetic modules are defined over GF(2^4), represented with the irreducible polynomial $p(y) = y^4 + y + 1$. The overall inversion takes place over GF(4^2), represented with the irreducible polynomial $q(w) = w^2 + w + 9$. However, passing from GF(2^8) to the composite field is not just byte-separation. It also requires the input to be multiplied with an isomorphic transform matrix, which requires the inverse transform prior to the output. Formation of the isomorphic transform matrix is explained in detail in [52].

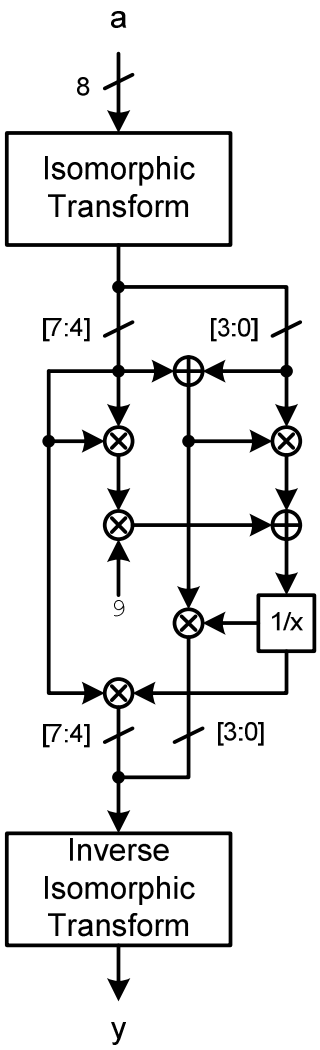

Figure 4.14 GF inverter module.

4.1.5 MixColumns Module

The module I/O and schematics are shown in Figure 4.15. It should be noted that all the multipliers and adders in the figure work over GF(2^8).

4.1.6 ShiftRows Module

The ShiftRows module is a look-up table of the state bytes. The look-up table formed from the shifted version of the bytes. Figure 4.16 shows the ShiftRows module.

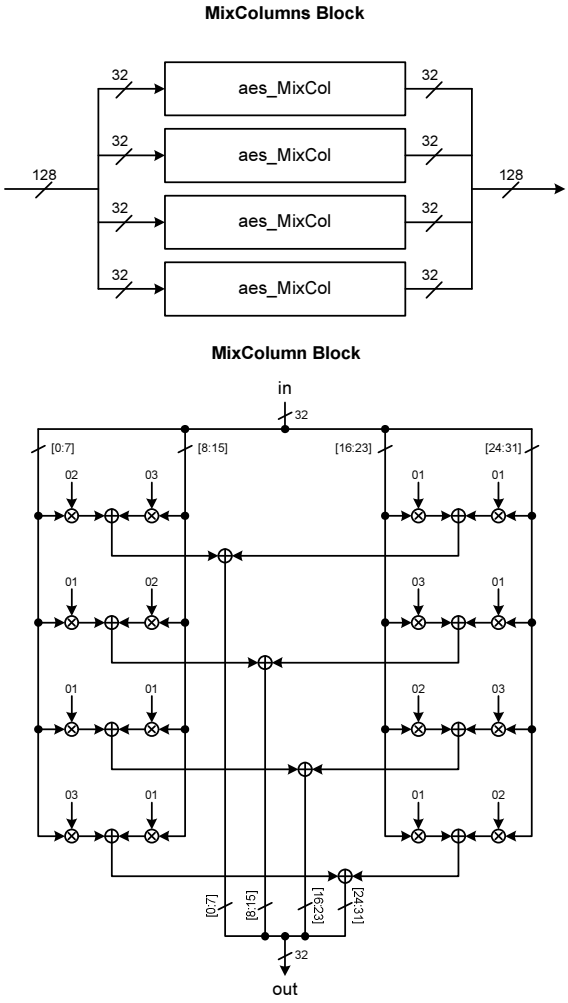

Figure 4.15 MixColumns and MixColumn modules.

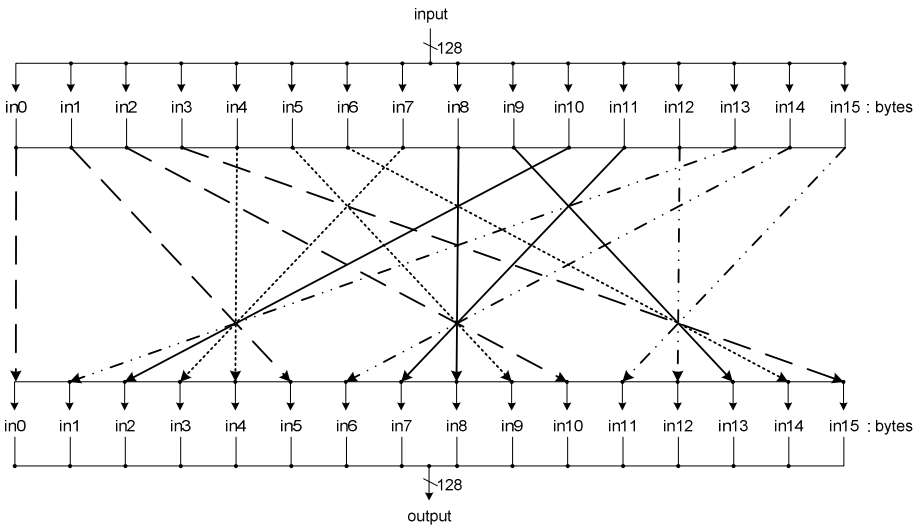

Figure 4.16 ShiftRows module.

4.1.7 Implementation Results

AES core is implemented on the smallest XILINX Virtex-5 device. The slice count is 1352 at a frequency of 67.3 MHz. Total cycle count for the processing of a 128-bit data block is 15 cycles, resulting in a throughput of 615.3 Mbps. The corresponding throughput/area number is 0.46 Mbps/slice. These results are compared against other AES implementations in Table 4.1.

Table 4.1 AES core implementation results.

Device	Freq (MHz)	Clock cycles	Block size (bits)	Area (slices)	Number of RAM Blocks	T/put (Mbps)	T/put / area (Mbps/slice)
xc5vlx30-3	67.3	14	128	1352	0	615.3	0.46
Virtex-E [27]	94.7	60	128	696	4	202	0.29
xcv1000 [53]	27.6	10	128	5673	0	353	0.06

4.2 Secure Hash Algorithm – 1 (SHA-1) Coprocessor

4.2.1 SHA-1 Algorithm

The SHA hash functions are a set of cryptographic hash functions designed by the National Security Agency (NSA) and published by the National Institute of Standards and Technology (NIST) as a U.S. Federal Information Processing Standard (FIPS) [21]. SHA stands for Secure Hash Algorithm. The five algorithms are denoted SHA-1, SHA-224, SHA-256, SHA-384, and SHA-512. The latter four variants are sometimes collectively referred to as SHA-2. SHA-1 is the most used function of the existing SHA hash functions and it is employed in several widely-used security applications and protocols. In 2005, security flaws were identified in SHA-1, namely that a mathematical weakness might exist, indicating that a stronger hash function would be desirable [54]. Although no successful attacks have yet been reported on the SHA-2 variants, they are algorithmically similar to SHA-1, so there have been efforts to develop improved alternative hash functions. As a result of this, a new hash standard, SHA-3, is currently under development for an ongoing NIST hash function competition which is scheduled to end with the selection of a winning function in 2012.

SHA-1 produces a message digest that is 160 bits long (and the numbers in the other four algorithms' names denote the bit length of the digest they produce). The following steps are utilized in the operation of SHA-1 algorithm:

- Preprocessing:
  - Padding the message,
  - Parsing the padded message into fixed-size blocks,
  - Setting the initial hash value.
- Hash computation (for each fixed-size message block):
  - Preparation of the message schedule,
  - Initialization of working variables,
  - Iterative calculation of internal hash values,
  - Computing the block hash value using the final internal and the previous block hash values.

Setting the resultant message digest to the final block hash value.

4.2.1.1 Description of the Function

SHA-1 may be used to hash a message, *M*, having a maximum length of 2^{64}-1 bits. The algorithm uses a message schedule of eighty 32-bit words, five working variables of 32 bits each, and a hash value of five 32-bit words. The final result of SHA-1 is a 160-bit message digest.

The words of the message schedule are labeled $W_0, W_1, \ldots, W_{79}$. The five working variables are labeled *A*, *B*, *C*, *D*, and *E*. The words of the hash value are labeled $H_0^{(i)}, H_1^{(i)}, \ldots, H_5^{(i)}$, which will hold the initial hash value, $H^{(0)}$, replaced by each successive intermediate hash value (after each message block is processed), $H^{(i)}$, and ending with the final hash value, $H^{(N)}$. SHA-1 also uses a single temporary word, T.

- Preprocessing:
  - The m-bit message, *M*, is first padded in order to ensure that the padded message length is a multiple of 512-bits. The padding is accomplished by appending the bit "1" to the end of the message, followed by k zero bits, where k is the smallest, non-negative solution to the equation $m+1+k \equiv 448 \bmod 512$.
  - The padded message is then parsed into Nx512-bit blocks, $M^{(1)}, M^{(2)}, \ldots, M^{(N)}$. The first 32-bits of message block *i* is denoted $M_0^{(i)}$, the next 32-bits $M_1^{(i)}$, and so on up to $M_{15}^{(i)}$.
  - The initial hash value is set as $H^{(0)} = H_0^{(0)} \| H_1^{(0)} \| H_2^{(0)} \| H_3^{(0)} \| H_4^{(0)}$, where each $H_i^{(0)}$ is a 32-bit hexadecimal word.

- Hash computation: Each message block, $M^{(i)}, i = 1, \ldots, N$, is processed in order to use the following steps:
  - Prepare the message schedule, W_t:

$$W_t = \begin{cases} M_t^{(i)} & 0 \le t \le 15 \\ ROTL^1(W_{t-3} \oplus W_{t-8} \oplus W_{t-14} \oplus W_{t-16}) & 16 \le t \le 79 \end{cases} \quad (4.4)$$

  - Initialize five working variables, *A*, *B*, *C*, *D*, and *E*, with the $(i-1)^{th}$ hash value:

$$\begin{aligned} A &= H_0^{(i-1)} \\ B &= H_1^{(i-1)} \\ C &= H_2^{(i-1)} \\ D &= H_3^{(i-1)} \\ E &= H_4^{(i-1)} \end{aligned} \quad (4.5)$$

  - For $t = 0, \ldots, 79$:

$$\begin{aligned} T &= ROTL^5(A) + f_t(B,C,D) + E + K_t + W_t \\ E &= D \\ D &= C \\ C &= ROTL^{30}(B) \\ B &= A \\ A &= T \end{aligned} \quad (4.6)$$

  - Compute the i^{th} intermediate hash value:

$$\begin{aligned} H_0^{(i)} &= A + H_0^{(i-1)} \\ H_1^{(i)} &= B + H_1^{(i-1)} \\ H_2^{(i)} &= C + H_2^{(i-1)} \\ H_3^{(i)} &= D + H_3^{(i-1)} \\ H_4^{(i)} &= E + H_4^{(i-1)} \end{aligned} \quad (4.7)$$

- After all the N 512-bit blocks are processed, the 160-bit resultant message digest is:

$$H_0^{(N)} \| H_1^{(N)} \| H_2^{(N)} \| H_3^{(N)} \| H_4^{(N)}. \quad (4.8)$$

4.2.1.2 SHA Functions

Here, the functions used by SHA-1 algorithm are defined. In addition to algorithm-specific functions, the common functions of all SHA functions are also defined.

4.2.1.2.1 Common Functions

The following functions are used by all SHA algorithms, either independently or within an algorithm-specific function:

- Logical operators:
  - $\wedge$: Bitwise logical AND,
  - $\vee$: Bitwise logical OR,
  - $\oplus$: Bitwise logical exclusive-OR,
  - $\neg$: Bitwise logical inversion.
- Addition modulo 2^w: The operation $x + y$ is defined as follows. The word *x* and *y* represent integers *X* and *Y*, where $0 \le X < 2^w$ and $0 \le Y < 2^w$. Compute $Z = (X + Y) \bmod 2^w$. Then $0 \le Z < 2^w$. Convert the integer *Z* to a word, *z*, and define $z = x + y$.
- Right shift operation $SHR^n(x)$, where *x* is a w-bit word and *n* is an integer with $0 \le n < w$, is defined by:

$$SHR^n(x) = x \gg n. \tag{4.9}$$

.

- Rotate right (circular right shift) operation $ROTR^n(x)$, where *x* is a w-bit word and *n* is an integer with $0 \le n < w$, is defined by:

$$ROTR^n(x) = (x \gg n) \vee (x \ll w - n). \tag{4.10}$$

- Rotate left (circular left shift) operation $ROTL^n(x)$, where *x* is a w-bit word and *n* is an integer with $0 \le n < w$, is defined by:

$$ROTL^n(x) = (x \ll n) \vee (x \gg w - n). \tag{4.11}$$

4.2.1.2.2 SHA-1 Functions

SHA-1 uses a sequence of logical functions, $f_0, f_1, \ldots, f_{79}$. Each function f_t, where $0 \le t \le 79$, operates on three 32-bit words, *x*, *y*, and *z*, and produces a 32-bit word as output. The function $f_t(x, y, z)$ is defined as follows:

$$f_t(x,y,z) = \begin{cases} Ch(x,y,z) = (x \wedge y) \oplus (\neg x \wedge z) & 0 \le t \le 19 \\ Parity(x,y,z) = x \oplus y \oplus z & 20 \le t \le 39 \\ Maj(x,y,z) = (x \wedge y) \oplus (x \wedge z) \oplus (y \wedge z) & 40 \le t \le 59 \\ Parity(x,y,z) = x \oplus y \oplus z & 60 \le t \le 79 \end{cases} \quad (4.12)$$

4.2.1.3 SHA-1 Constants

SHA-1 uses a sequence of eighty constant 32-bit words, $K_0, K_1, \ldots, K_{79}$, which are given in hexadecimal as:

$$K_t = \begin{cases} \mathbf{5a827999} & 0 \le t \le 19 \\ \mathbf{6ed9eba1} & 20 \le t \le 39 \\ \mathbf{8f1bbcdc} & 40 \le t \le 59 \\ \mathbf{ca62c1d6} & 60 \le t \le 79 \end{cases} \quad (4.13)$$

4.2.1.4 SHA-1 Initial Hash Values

For SHA-1, the initial hash value, $H^{(0)}$, shall consist of the following five 32-bit words, in hexadecimal as:

$$\begin{aligned} H_0^{(0)} &= \mathbf{67452301} \\ H_1^{(0)} &= \mathbf{efcdab89} \\ H_2^{(0)} &= \mathbf{98badcfe} \\ H_3^{(0)} &= \mathbf{10325476} \\ H_4^{(0)} &= \mathbf{c3d2e1f0} \end{aligned} \quad (4.14)$$

4.2.2 Architecture Overview

The coprocessor handles the SHA-1 hashing. Preprocessing is implemented by the microprocessor software. Upon completion of preprocessing, microprocessor passes data to the SHA-1 coprocessor in 512-bit blocks. Each 512-bit block corresponds to a 16-word message, and

is written into the SHA-1 input registers. SHA-1 coprocessor starts hashing via the “start” pulse which goes high when the microprocessor writes into the virtual SHA-1 command/status register. Once the hashing is completed, the microprocessor can either pass a new 512-bit input block or read the 160-bit hash output from the SHA-1 output registers. Prior to the processing of the first input block, SHA-1 coprocessor internal states have to be cleared via the clear signal.

The pseudo-code for the SHA-1 hashing is given below:

```
SHA1 (word in[16], word hash[5], word W[80],
      word  K[80], word A,B,C,D,E,T)

if clear = 1 then
    hash[0] = hash_init[0]
    hash[1] = hash_init[1]
    hash[2] = hash_init[2]
    hash[3] = hash_init[3]
    hash[4] = hash_init[4]
else
    begin
        A = hash[0]
        B = hash[1]
        C = hash[2]
        D = hash[3]
        E = hash[4]
        for t = 0 step 1 to 79
            if round < 16 then
                W[t] = in[t]
            else
                W[t] = rotl_1( W[t-3] ^ W[t-8] ^
                               W[t-14] ^ W[t-16] )
            end if
            T = rotl_5(A) + f_t(B,C,D) + E + K[t] + W[t]
            E = D
            D = C
            C = rotl_30(B)
            B = A
            A = T
        end for
        hash[0] = A
        hash[1] = B
        hash[2] = C
        hash[3] = D
        hash[4] = E
    end

end if
```

The SHA-1 coprocessor is basically a complex state machine, which implements the SHA-1 algorithm explained in detail in the preceding subsections. During a single hashing operation, the SHA-1 functional loop has to be iteration for a total of 80 times. This directly maps to a state

machine which combines the input with its internal state, and then runs the internal state through its combinational path for 80 cycles before accepting new input.

As in the case of AES, new input is registered with the "start" pulse, which also activates the SHA-1 state machine. A total of 80-words of message digest are generated from the 16-word input. At each cycle of the state machine run, the message digest word corresponding to that cycle is combined with the present state in that cycle through a combination of functions in order to generate the next state. At the end of all 80 cycles, words of the internal state are added individually with the words of the stored hash output. The hash output is also another state variable, but updated only once at the end of all 80 cycles. Its initial value is a 160-bit constant, which is loaded with the "clear" pulse.

The first 16 words of the message digest are the input message words, which are shifted into a shift register of width 32-bits, and depth 16. After the first 16 values, the shift register is operated as a feedback shift register, which generates its next input word from a combination of its internal words. This is done for another 64 cycles until all 80 words of the message digest.

The message digest generation is run in parallel with the state machine's hashing cycles. This way, hardware parallelism is fully exploited and no cycles are lost. The generic state machine using the "initialization, iteration, finalization" phases model explained before is also applied here, resulting in the data flow given below:

```
1. Initialization (start comes)
    act     ← 1      :active status
    cnt     ← 0      :state counter
    w       ← input
    s       ← hash
    w_out   =  w[0:31]
    w_inp   =  ROTL^1 (w[0:31]^ w[64:95]^ w[256:287]^ w[416:447])
    w_next  =  w[32:511] || w_inp
    A_next  =  ROTL^5(A) + f(cnt)(B,C,D) + E + K(cnt) + w_out
    B_next  =  A
    C_next  =  ROTL^30(B)
    D_next  =  C
    E_next  =  D
    s_next  =  A_next || B_next || C_next || D_next || E_next

2. Iteration (cnt=1 to 79)
    cnt     ← cnt + 1
    Sreg    ← state
    Kreg    ← key
    Rcon    ← rc
    key     =  KeyRound[Kreg, Rcon]
    rc      =  Rcon x 2
    state   =  MixColumns{ShiftRows[SubBytes(Sreg)]} ⊕ key
```

```
    cnt     ← cnt+1
    w       ← w_next
    s       ← s_next
    w_out   ← w[0:31]
    w_inp   =  ROTL^1 (w[0:31]^ w[64:95]^ w[256:287]^ w[416:447])
    w_next  =  w[32:511] || w_inp
    A_next  =  ROTL^5(A) + f(cnt)(B,C,D) + E + K(cnt) + w_out
    B_next  =  A
    C_next  =  ROTL^30(B)
    D_next  =  C
    E_next  =  D
    h_next  =  hash + s

3. Finalization (cnt=80)
    act     ← 0
    cnt     ← 80
    hash    ← h_next
    output  =  hash

← : Sequential
=  : Combinational
|| : Concatenation
```

The active signal and counter are parts of the control module, which generates all control signals that organize the data traffic between the state registers and combinational blocks. Figure 4.17 shows the block diagram for the SHA-1 coprocessor core.

This core is then put into the SHA-1 wrapper, which provides the RAM-like behavior of the SHA-1 coprocessor. From the microprocessor's point of view, SHA-1 input, output and configuration registers are just addresses inside the memory map: The input and configuration registers are write-only addresses, while the output registers are read-only. There is no access to the internal state registers for the microprocessor. This scheme is shown in Figure 4.18.

The timing diagram of the SHA-1 block can be seen in Figure 4.19.

4.2.3 Message Scheduler

Message schedule schematic for SHA-1 is shown in Figure 4.20. It should be noted that the most significant 32-bit word of the 512-bit message block is M_0 while the least significant word is M_{15} and $0 \le t \le 79$.

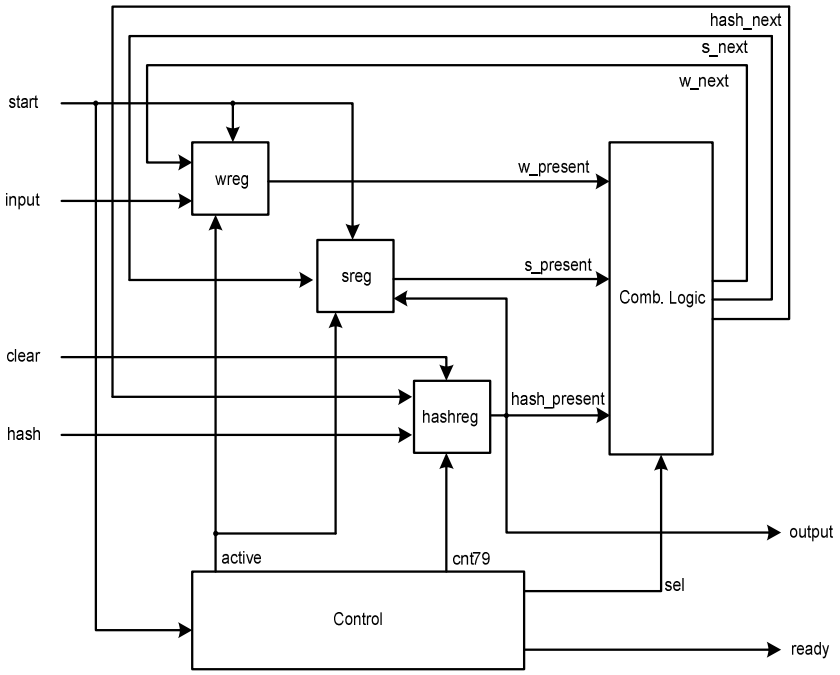

Figure 4.17 SHA-1 coprocessor core block diagram.

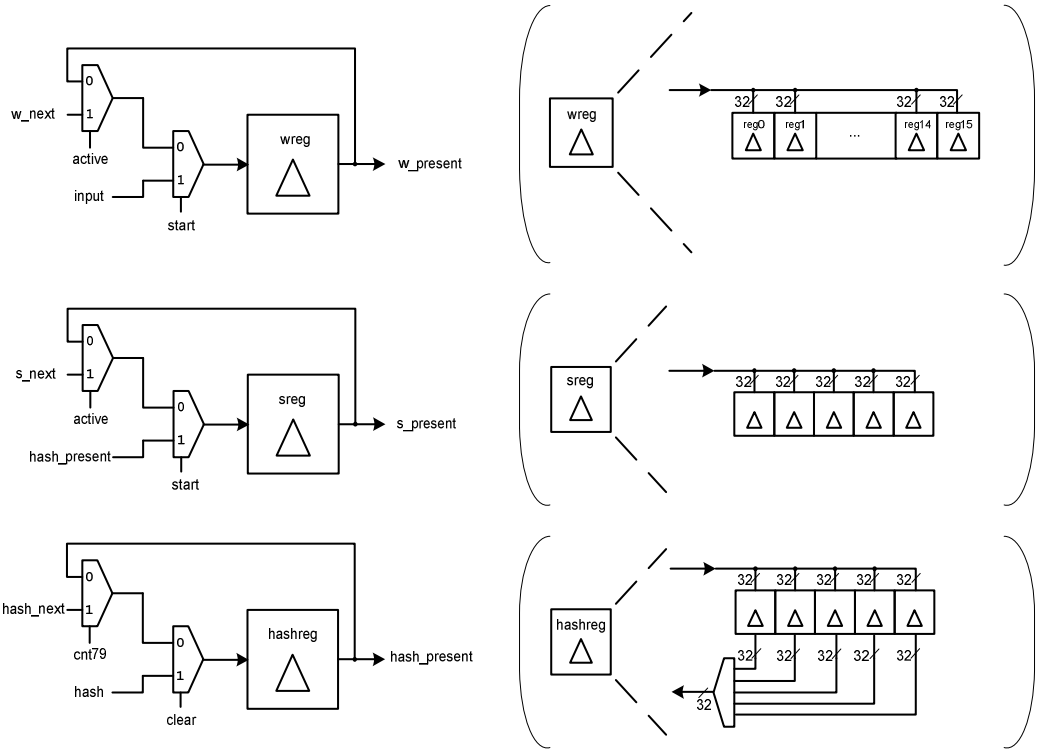

Figure 4.18 SHA-1 register and wrapper details.

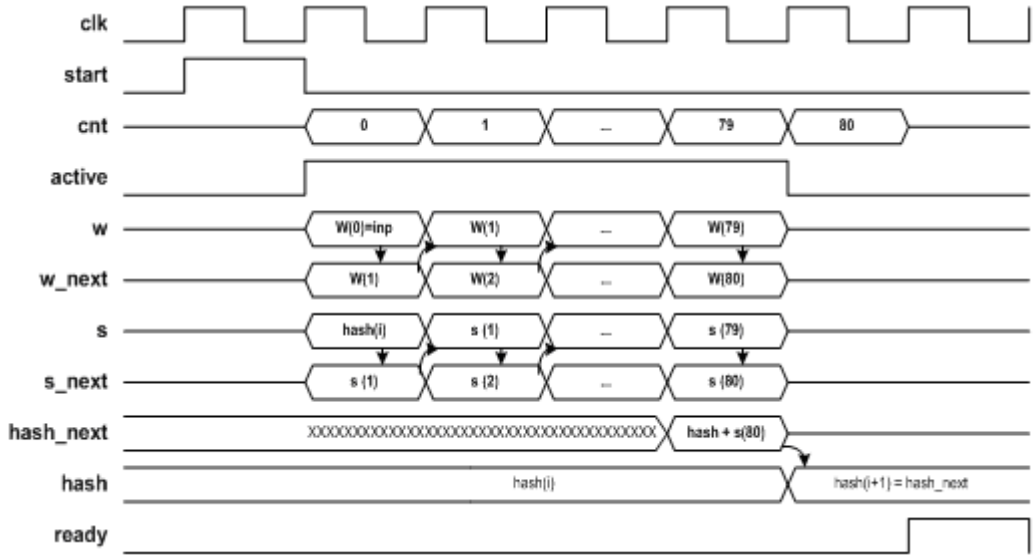

Figure 4.19 Timing diagram of SHA-1 hardware block.

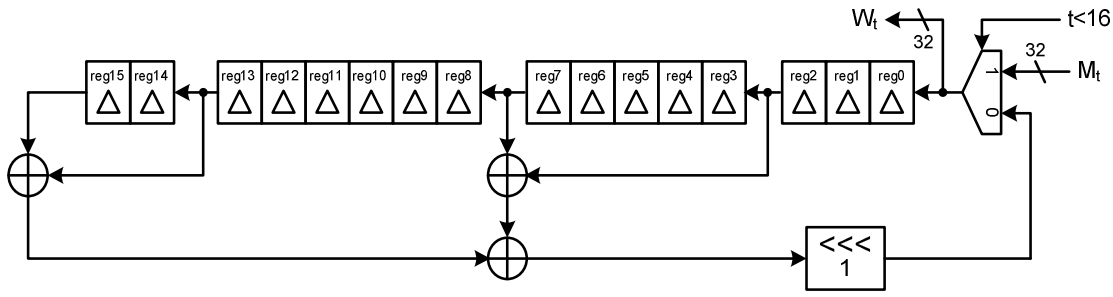

Figure 4.20 Message scheduler schematic.

4.2.4 Round Function

Round function schematic for SHA-1 is shown in Figure 4.21. It should be noted that the most significant 32-bit word of the 160-bit hash value is *A* while the least significant word is *E*.

4.2.5 Implementation Results

SHA-1 core is implemented on the smallest XILINX Virtex-5 device. The slice count is 342 at a frequency of 127.7 MHz. Total cycle count for the processing of a 512-bit data block is 80 cycles, resulting in a throughput of 817.3. The corresponding throughput/area number is 2.39 Mbps/slice. Table 4.2 summarizes the results together with the figures from a reference implementation.

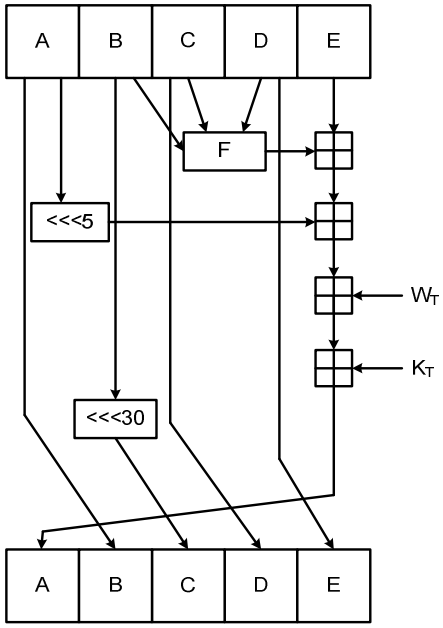

Figure 4.21 Round function block diagram.

Table 4.2 SHA-1 core implementation results.

Device	Freq (MHz)	Clock cycles	Block size (bits)	Area (slices)	T/put (Mbps)	T/put / area (Mbps/slices)
xc5vlx30-3	127.7	80	512	342	817.3	2.39
xcv-1000-6 [55]	72.2	80	512	1475	462	0.31

4.3 Montgomery Modular Multiplier (MMM) Coprocessor

4.3.1 MMM Algorithm

RSA algorithm [19] is one of the simplest public-key cryptosystems in terms of mathematical complexity. It is based on the modular exponentiation of the input message. The exponent used in the encryption process is the public key, whereas the exponent used in the decryption process is the private key.

RSA encryption and decryption operations are defined as:

$$c = m^e \bmod n \tag{4.15}$$

and

$$m = c^d \bmod n \tag{4.16}$$

respectively, where

- m: N-bit message (plaintext) , an integer between 0 and n-1
- c: N-bit ciphertext, an integer between 0 and n-1
- n: RSA modulus, an N-bit positive integer that is a product of 2 distinct odd primes
- e: RSA public key, a E-bit positive integer ($E << N$)
- d: N-bit private key

Modular exponentiation operation, which RSA is based on, is in turn based on modular multiplication. Assuming that all x, z, w are N-bit binary numbers and y is an E-bit binary number, i.e.

$$x = x_{N-1}x_{N-2}\ldots x_0\,, \tag{4.17}$$

$$y = y_{E-1}y_{E-2}\cdots y_0\,, \tag{4.18}$$

$$z = z_{N-1}z_{N-2}\ldots z_0\,, \tag{4.19}$$

$$w = w_{N-1}w_{N-2}\ldots w_0\,, \tag{4.20}$$

and

$$w = x^y \bmod z\,, \tag{4.21}$$

it can be calculated using the algorithm below:

$$\begin{aligned}
&M_0 = x,\ R_0 = 1\\
&\text{for } i = 0 \text{ to } E-1\\
&\quad R_{i+1} = \begin{cases} R_i \times M_i \bmod z & \text{if } y_i = 1\\ R_i & \text{else } (y_i = 0) \end{cases}\\
&\quad M_{i+1} = M_i \times M_i \bmod z\\
&\text{end for}\\
&w = R_E
\end{aligned} \tag{4.22}$$

Now, a modular multiplication operator MM( ) can be defined, where

$$\mathrm{MM}(a,b,c) = a \times b \bmod c\,, \tag{4.23}$$

and the given algorithm can be rewritten by substituting this operator:

$$\begin{aligned}
&M_0 = x,\ R_0 = 1 \\
&\text{for } i = 0 \text{ to } E-1 \\
&\quad R_{i+1} = \begin{cases} \mathrm{MM}(R_i, M_i, z) & \text{if } y_i = 1 \\ R_i & \text{else} \end{cases} \\
&\quad M_{i+1} = \mathrm{MM}(M_i, M_i, z) \\
&\text{end for} \\
&w = R_E
\end{aligned} \tag{4.24}$$

As seen, modular exponentiation can easily be implemented as a series of modular multiplications, provided that there exists an ideal modular multiplication operator MM( ). However, in practice, it is costly to implement an ideal modular multiplication module in hardware. Instead, what is known as Montgomery multiplication algorithm [33-36] is preferred for hardware implementation. This algorithm is defined as:

$$\begin{aligned}
&S_0 = 0 \\
&\text{for } i = 0 \text{ to } N-1 \\
&\quad q_i = (S_i + a_i b) \bmod 2 \\
&\quad S_{i+1} = (S_i + q_i c + a_i b)/2 \\
&\text{end for} \\
&\text{if } S_N \geq c \\
&\quad d = S_N - c \\
&\text{else} \\
&\quad d = S_N \\
&\text{end if}
\end{aligned} \tag{4.25}$$

where

$$d = \mathrm{MMM}(a,b,c) = a \times b \times 2^{-N} \bmod c\,. \tag{4.26}$$

Montgomery modular multiplication algorithm is very simple in nature, and suitable for digital implementation. It only requires 1-bit multiplication, which maps to logical AND operation, and *N*-bit addition, for which plenty of optimization possibilities exist. However, it has two drawbacks in its simplest form:

- It takes N +1 cycles to complete using two N-bit adders. Adder optimization is possible via time-sharing a single adder for each addition within the for-loop. In this case, the whole algorithm takes $2N$+1 cycles to complete. In this implementation, this drawback could be ignored as the compactness of the core is more important than the cycle count.
- Montgomery modular multiplication introduces an extra and undesired 2^{-N} factor into the multiplication result. This extra factor has to be taken care of, by modifying the original modular exponentiation algorithm, which leads to the Montgomery modular exponentiation.

Montgomery modular exponentiation algorithm is defined as:

$$
\begin{aligned}
&k \equiv 2^{2N} \bmod z \\
&M_0 = \mathrm{MMM}(x, k, z) \\
&R_0 = \mathrm{MMM}(1, k, z) \\
&\text{for } \ i = 0 \ \text{ to } \ E-1 \\
&\quad R_{i+1} = \begin{cases} \mathrm{MMM}(R_i, M_i, z) & \text{if } \ y_i = 1 \\ R_i & \text{else} \end{cases} \\
&\quad M_{i+1} = \mathrm{MMM}(M_i, M_i, z) \\
&\text{end for} \\
&w = \mathrm{MMM}(R_E, 1, z)
\end{aligned}
\tag{4.27}
$$

where

$$
w = x^y \bmod z\,. \tag{4.28}
$$

It should be noticed that this algorithm requires pre-calculation of $k = 2^{2N} \bmod n$ as an additional task. However, since the modulus seldom changes during RSA operation, this calculation is done once in a while and k is supplied to the algorithm as a constant.

4.3.2 Architecture Overview

As explained in the algorithm section, Montgomery modular multiplication operations requires addition of N-bit numbers, where N can be as big as 2048-bits. Performing such large multiplications in a single cycle is neither feasible, nor practical. It is logical to split the input data into smaller portions and perform the addition on these portions in each cycle while propagating the carry output result from each addition to the next cycle as the carry input. This way, addition time increases from a single cycle to several cycles, while the hardware complexity diminishes enormously.

The data width of these portions is an important parameter, which determines both the execution time of the algorithm and the hardware complexity of the core. It has to be chosen very carefully. Luckily, the 32-bit bus width of the processor presents a perfect choice. The data portion width can also be selected as 32-bits. This way, adder input words can be directly read from separate RAMs in parallel in each cycle, addition is performed on these words, the addition result word is written to the result RAM, and the 1-bit carry output is propagated to the next cycle as the carry input.

In terms of the algorithm's pseudo code given before, each addition is now replaced with a for-loop iterated over all words of the inputs. For the case of 512-bits, a single addition takes a total of 512/32=16 cycles, while for the worst case of 2048-btis a single addition takes a total of 2048/32=64 cycles.

However, in the actual algorithm, the main state update is not performed by just the addition of two very long numbers. Instead, three long numbers are added (one of them being the present value of the state), the result is divided by 2 (shifted right by 1-bit), and sent to the state register as the next state.

The partial addition scheme can still be applied: Instead of a 32-bit adder with two inputs, a 32-bit adder with three inputs is used. The right shifting is implementing by taking the least significant 1-bit of the current addition result and concatenating it with the most significant 31-bits of the previous addition result. The resultant 32-bit word corresponds to the addition-followed-by-division result word of the previous cycle. Therefore, the final result will be completed with 1 cycle of pipeline delay.

The whole scheme and application of it to the MMM algorithm can be best explained by means of a scaled-down example, whose pseudo-code is given below. In this example, the total data width is assumed to be 16-bits (instead of the actual 512 to 2048 bits), while the width of each data word is only 4-bits (instead of the actual 32-bits). This means that each 16-bit addition will be completed in 4 cycles.

It should also be noted that instead of the iteration phase in AES or SHA-1, there is the phase-0, which is divided into multiple loops and cycles. Each loop corresponds to the multiplication of the multiplicand input with a bit of the multiplier input, resulting in a total of 16 loops for this specific example. Each cycle corresponds to a partial addition cycle, which corresponds to a total of 4 cycles per loop for this example.

Furthermore, instead of the finalization phase in AES or SHA-1, there are the phase-1 and phase-2, where the subtraction of the modulus from the state and re-addition onto the state are implemented, respectively. If the subtraction result of phase-1 is positive, there is no need for re-addition. The state machine execution is terminated. Both of these phases are implemented in 4-cycles because of partial addition structure.

```
Initialization:
    Sext  =  0 ;  S  =  S0 || S1 || S2 || S3  =  0 ;

Phase 0:

    Loop 0:
        Atmp  ← A3
        q     =  S3(0) ^ ( Atmp(0)&B3(0) )

        Cycle 0:
            t[6:0]  =  S3 + (q?N3:0) + (a0?B3:0)
            tmp     ← t[3:1]
            cout    ← t[5:4]

        Cycle 1:
            t[6:0]  =  S2 + (q?N2:0) + (a0?B2:0) + cout
            tmp     ← t[3:1]
            cout    ← t[5:4]
            S3      ← {t[0],tmp}

        Cycle 2:
            t[6:0]  =  S1 + (q?N1:0) + (a0?B1:0) + cout
            tmp     ← t[3:1]
            cout    ← t[5:4]
            S2      ← {t[0],tmp}

        Cycle 3:
            t[6:0]=S0 + (q?N0:0) + (a0?B0:0) + cout + {sext,0000}
            tmp     ← t[3:1]
            S1      ← {t[0],tmp}
            t[0]    ← t[4]
            sext    ← t[6:5]

    Loop 1:
        Atmp  ← Atmp >> 1
        q     =  S3(0) ^ ( Atmp(0)&B3(0) )

        Cycle 0:
            S0      ← {t[0],tmp}
            t[6:0]  =  S3 + (q?N3:0) + (a0?B3:0)
            tmp     ← t[3:1]
            cout    ← t[5:4]

        Cycle 1:
            t[6:0]  =  S2 + (q?N2:0) + (a0?B2:0) + cout
            tmp     ← t[3:1]
            cout    ← t[5:4]
```

```
            S3      ← {t[0],tmp}

        Cycle 2:
            t[6:0]  =  S1 + (q?N1:0) + (a0?B1:0) + cout
            tmp     ← t[3:1]
            cout    ← t[5:4]
            S2      ← {t[0],tmp}

        Cycle 3:
            t[6:0]=S0 + (q?N0:0) + (a0?B0:0) + cout + {sext,0000}
            tmp     ← t[3:1]
            S1      ← {t[0],tmp}
            t[0]    ← t[4]
            sext    ← t[6:5]

    Loop 2:
        Atmp  ← Atmp >> 1
        q     =  S3(0) ^ ( Atmp(0)&B3(0) )

        Cycle 0:
            S0      ← {t[0],tmp}
            t[6:0]  =  S3 + (q?N3:0) + (a0?B3:0)
            tmp     ← t[3:1]
            cout    ← t[5:4]

        Cycle 1:
            .
            .
            S3

        Cycle 2:
            .
            .
            S2

        Cycle 3:
            .
            .
            S1

    Loop 3:
        Atmp  ← Atmp >> 1
        q     =  S3(0) ^ ( Atmp(0)&B3(0) )

        Cycle 0:
            S0
            .
            .

        Cycle 1:
            .
            .
            S3

        Cycle 2:
            .
            .
            S2
```

```
        Cycle 3:
            .
            .
            S1

    Loop 4:
        Atmp  ← A2
        q     =  S3(0) ^ ( Atmp(0)&B3(0) )

        Cycle 0:
            S0
            .
            .

        Cycle 1:
            .
            .
            S3

        Cycle 2:
            .
            .
            S2

        Cycle 3:
            .
            .
            S1

    Loop 5:
        Atmp  ← Atmp >> 1
        q     =  S3(0) ^ ( Atmp(0)&B3(0) )

        Cycle 0:
            S0
            .
            .

        Cycle 1:
            .
            .
            S3

        Cycle 2:
            .
            .
            S2

        Cycle 3:
            .
            .
            S1

    Loop 6:
        Atmp  ← Atmp >> 1
        .
        .
        .

    Loop 7:
        Atmp  ← Atmp >> 1
```

```
        .
        .
        .

    Loop 8:
        Atmp  ← A1
        .
        .
        .

    Loop 9:
        Atmp  ← Atmp >> 1
        .
        .
        .

    Loop 10:
        Atmp  ← Atmp >> 1
        .
        .
        .

    Loop 11:
        Atmp  ← Atmp >> 1
        .
        .
        .

    Loop 12:
        Atmp  ← A1
        .
        .
        .

    Loop 13:
        Atmp  ← Atmp >> 1
        .
        .
        .

    Loop 14:
        Atmp  ← Atmp >> 1
        .
        .
        .

    Loop 15:
        Atmp  ← Atmp >> 1
        q     =  S3(0) ^ ( Atmp(0)&B3(0) )

        Cycle 0:
            S0      ← {t[0],tmp}
            t[6:0]  =  S3 + (q?N3:0) + (a0?B3:0)
            tmp     ← t[3:1]
            cout    ← t[5:4]

        Cycle 1:
            t[6:0]  =  S2 + (q?N2:0) + (a0?B2:0) + cout
            tmp     ← t[3:1]
```

```
            cout    ← t[5:4]
            S3      ← {t[0],tmp}

        Cycle 2:
            t[6:0]  =  S1 + (q?N1:0) + (a0?B1:0) + cout
            tmp     ← t[3:1]
            cout    ← t[5:4]
            S2      ← {t[0],tmp}

        Cycle 3:
            t[6:0]=S0 + (q?N0:0) + (a0?B0:0) + cout + {sext,0000}
            tmp     ← t[3:1]
            S1      ← {t[0],tmp}
            t[0]    ← t[4]
            sext    ← t[6:5]


Phase 1:

    Cycle 0:
        S0      ← {tmp, t[0]}
        t[6:0]  =  S3 + ~N3 + cout
        tmp     ← t[3:1]
        cout    ← t[5:4]

    Cycle 1:
        t[6:0]  =  S2 + ~N2 + cout
        tmp     ← t[3:1]
        cout    ← t[5:4]
        S3      ← {tmp, t[0]}

    Cycle 2:
        t[6:0]  =  S1 + ~N1 + cout
        tmp     ← t[3:1]
        cout    ← t[5:4]
        S2      ← {tmp, t[0]}

    Cycle 3:
        t[6:0]  =  S0 + ~N0 + {11,0000} + {sext,0000} + cout
        tmp     ← t[3:1]
        S1      ← {tmp, t[0]}
        t[0]    ← t[4]
        sext    ← t[6:5]
        sbit    =  t[5]
        term    =  ~sbit


Phase 2:

    Cycle 0:
        S0      ← {tmp, t[0]}
        t[6:0]  =  S3 + N3 + cout
        tmp     ← t[3:1]
        cout    ← t[5:4]
        if (term) stop
        else
```

```
    Cycle 1:
        t[6:0]  =  S2 + N2 + cout
        tmp     ← t[3:1]
        cout    ← t[5:4]
        S3      ← {tmp, t[0]}

    Cycle 2:
        t[6:0]  =  S1 + N1 + cout
        tmp     ← t[3:1]
        cout    ← t[5:4]
        S2      ← {tmp, t[0]}

    Cycle 3:
        t[6:0]  =  S0 + N0 + cout
        tmp     ← t[3:1]
        S1      ← {tmp, t[0]}
        t[0]    ← t[4]
        sext    ← t[6:5]

    Cycle X
        S0      ← {tmp, t[0]}



← : Sequential
=  : Combinational
```

The hardware block diagram of MMM is given in Figure 4.22. As seen from the figure, all RAM outputs, state extension bits and carry in bits are sent to adder (except *A_ram*, it is stored in *Atmp* register first, in order to shift the data when needed) via the control signals, and least significant bit of the result and the part stored in *tmp* is then sent to combiner (where the combination is done according to the phase). The combined result is then stored into *S_ram* and the operation continues so on. The output is then read from the *S_ram*. It should be noted that, for 32-bit memory address, the addition result bits which are stored in *tmp*, *carry_in* and *s_ext* will be different (i.e. 4-bit memory's t[3:1] will be t[31:1] in 32-bit memory address approach).

The timing diagram of the MMM block can be seen in Figure 4.23. The phase, loop, cycle and early termination signals given in the dataflow can be easily seen in this waveform. There exists one clock delay between the waveform and the dataflow because of the pipeline.

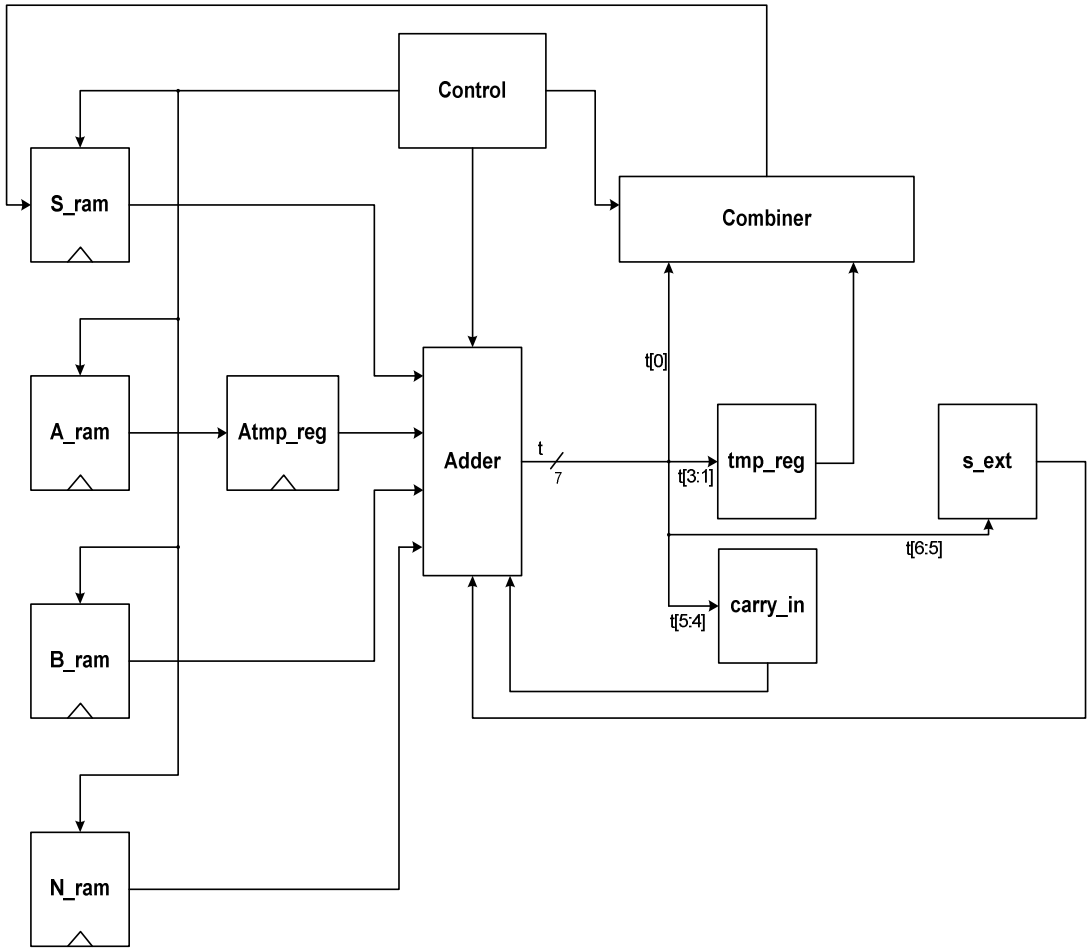

Figure 4.22 MMM block diagram.

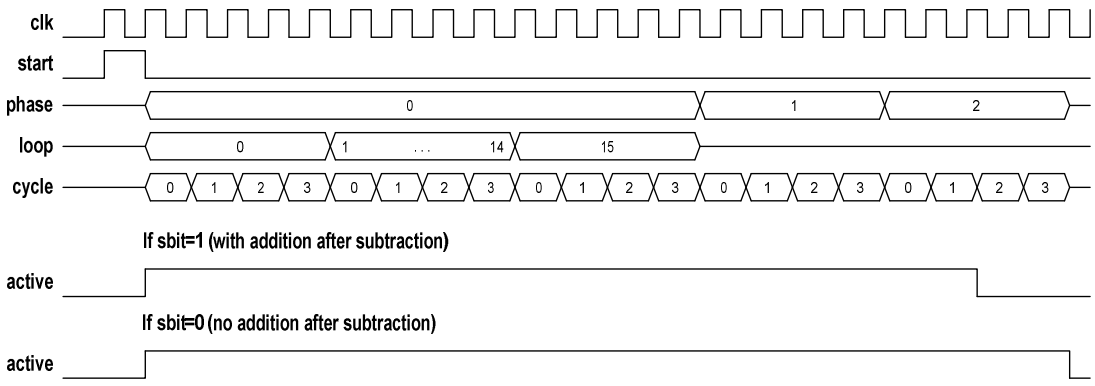

Figure 4.23 Timing diagram of MMM hardware block.

4.3.3 Word-serial Adder

The word-serial adder performs addition operation according to the control signals, which select the active inputs depending on the phase, loop, and cycle.

Phase 0 is the phase where the modular multiplication is performed. All inputs of the addition (S_{out}, N_{out}, B_{out}, carry_in, and s_ext in the last cycle) are selected for the operation. N_{out} and B_{out} are added in case that there exist non-zero q and a0 bits, respectively. Otherwise, N_{out} and B_{out} are masked.

The addition can again be explained for the 4-bit wide words: At first, the least significant 4 bits of the 16-bit S_{out}, N_{out} and B_{out} are added and the carry output is stored in the carry_in register. Then, the next 4 bits of these terms are taken for addition, together with the stored carry_in bits. This addition continues for 4 cycles, and then in the last cycle, s_ext bits are also taken into account. The overall result is formed by combining the 4-bit result of every cycle. This operation is shown in detail in Figure 4.24.

Phase 1 is the comparison part of the multiplication against the modulus (subtraction of N from multiplication result). S_{out}, N_{out}, carry_in, and s_ext (in the last cycle) terms are selected for the operation. To perform subtraction, the 2's complement of N_{out} is taken. This is implemented by adding the inverse of modulus to the multiplication result together with an initial carry_in value of 1, i.e. $S_{next} = S_{out} + \sim N_{out} + 1$.

Firstly, the least significant 4 bits of the 16-bit S_{out} and N_{out} are added and the carry output is stored in the carry_in register. Then, the next 4 bits of these terms are taken for addition, with stored carry_in bits. This addition continues for 4 cycles, and then in the last cycle, s_ext bits are also taken into account. The overall result is formed by the 4-bit result of every cycle. The operation (for 4-bit register case) can be seen in Figure 4.25, in detail.

Phase 2 is performed if re-addition of N_{out} is required (in case the result of phase 1 is negative). S_{out}, N_{out}, carry_in, and s_ext (in the last cycle) terms are selected for the operation.

At first, the least significant 4 bits of the 16-bit S_{out} and N_{out} are added and the carry output is stored in the carry_in register. Then, the next 4 bits of these terms are taken for addition, with stored carry_in bits. This addition continues for 4 cycles, and then in the last cycle, s_ext bits are also taken into the addition. The overall result is formed by the 4-bit result of every cycle. The operation (for 4-bit register case) can be seen in Figure 4.26, in detail.

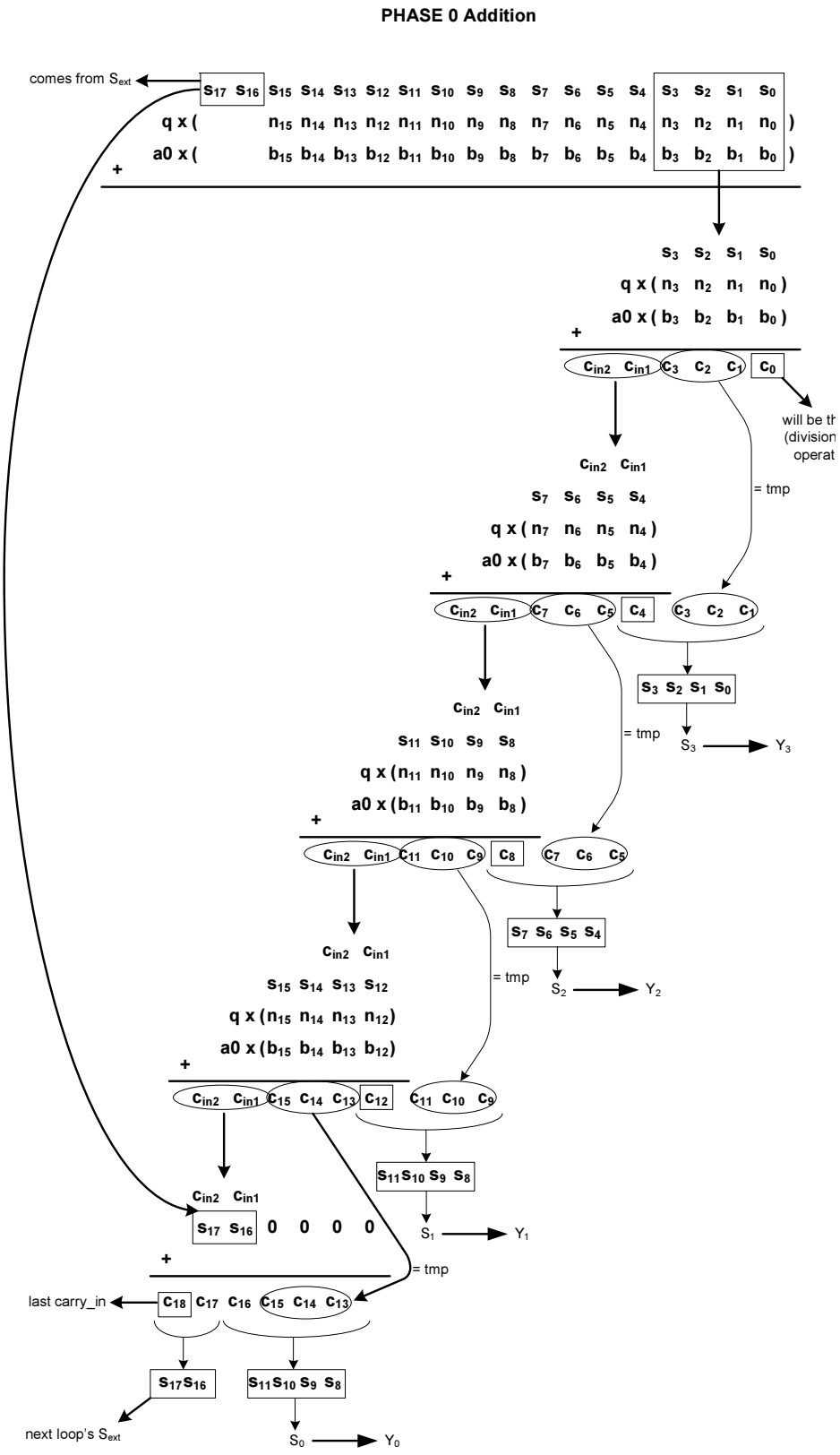

Figure 4.24 Addition in phase 0.

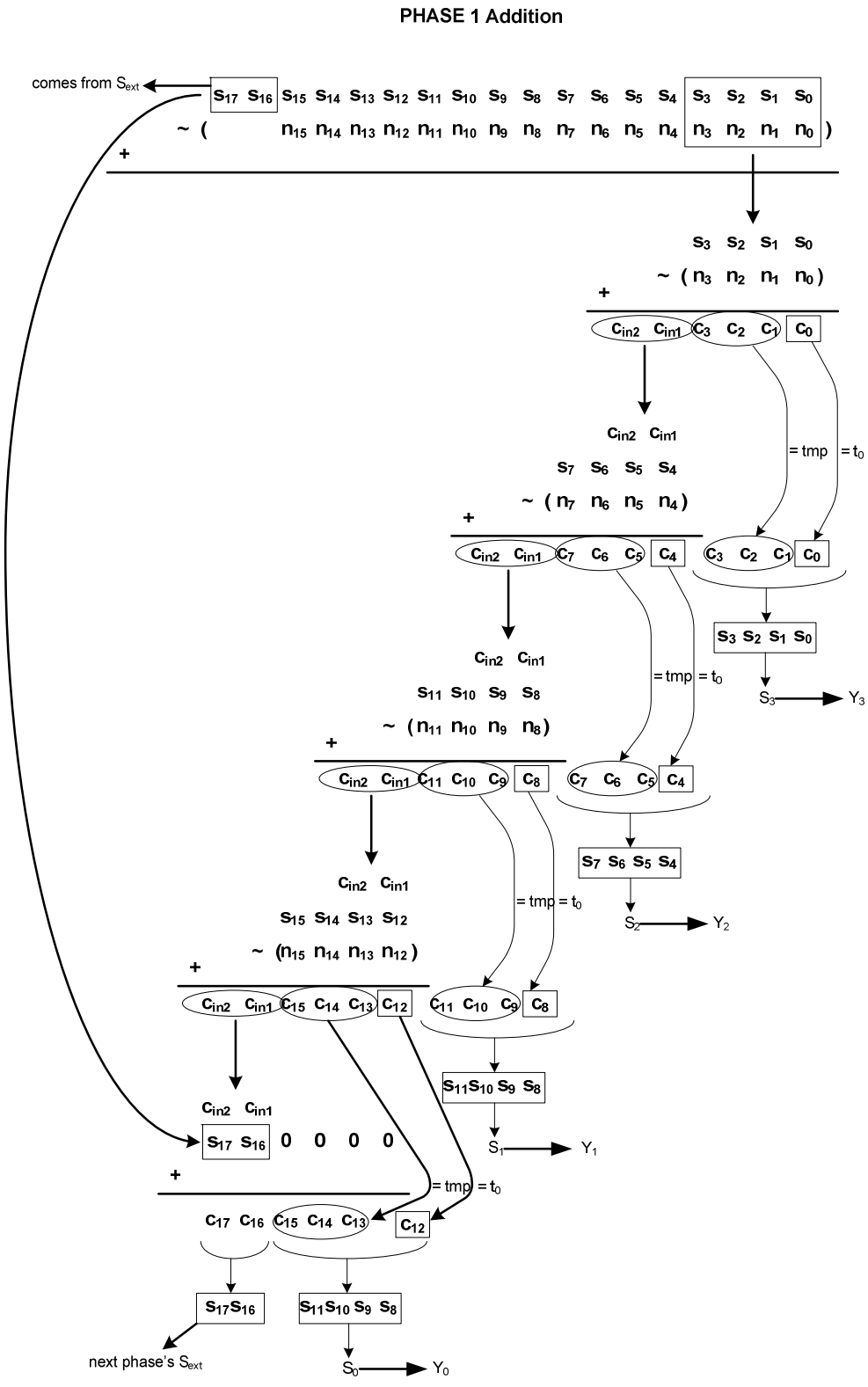

Figure 4.25 Subtraction in phase 1.

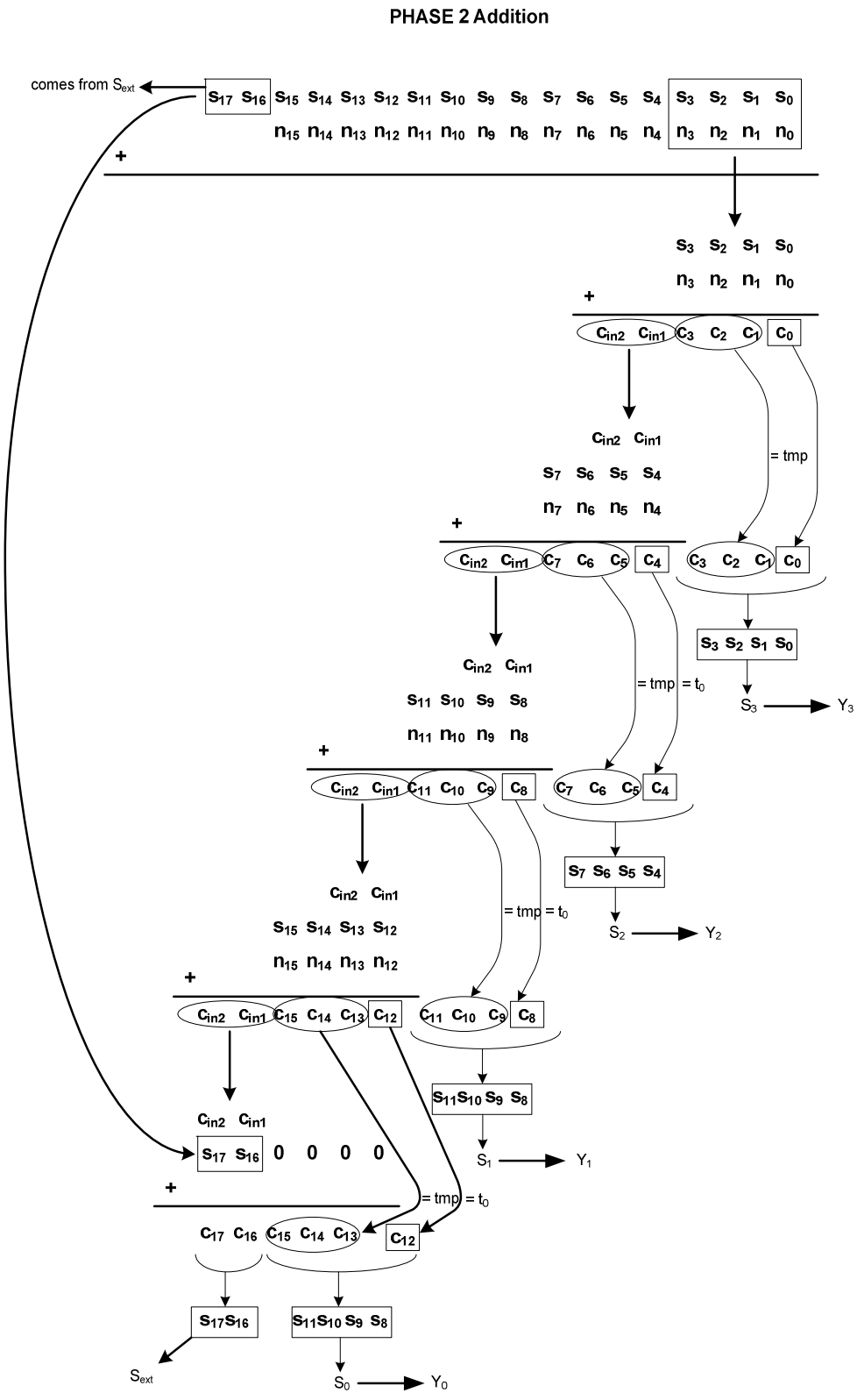

Figure 4.26 Addition in phase 2.

In Figure 4.27, the overall adder block is shown. The multiplexers select the inputs according to phase, loop and cycle information coming from the control logic. The selected signals are then summed up by the 4-input adder.

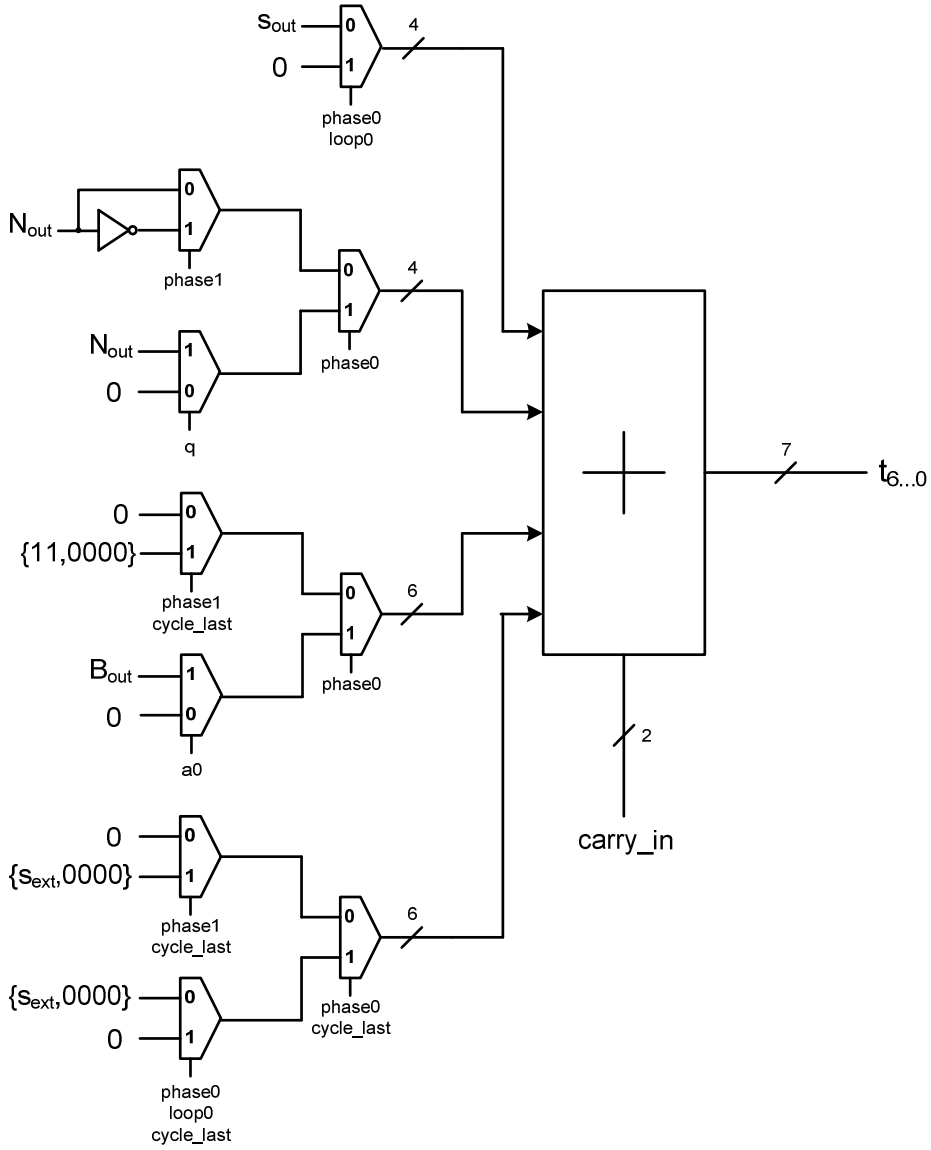

Figure 4.27 Adder block diagram.

4.3.4 Implementation Results

MMM core is implemented on the smallest XILINX Spartan device and smallest Virtex-5 device. With the Spartan family, the slice count is 224 at a frequency of 73.3 MHz. For the Virtex-5 family, the slice count is 101 at a frequency of 115.3 MHz.

For a fair comparison with existing designs, the throughput is calculated for 1024-bit RSA calculation using the MMM module. Including an estimated worst case 10 percent of software overhead, the total cycle count for the processing of a 1024-bit data block is about 55.5 million cycles. This results in a throughput of 1.4 Kbps for the Spartan device and 2.1 Kbps for the Virtex-5 device. The corresponding throughput/area numbers are 0.006 and 0.02 Kbps/slice, respectively.

Table 4.3 compares these results with reference designs. It should be noted that the slice counts for the other reference designs include the extra control logic around the modular multiplier cores required for the RSA operation.

Table 4.3. RSA core implementation results.

Device	**Freq (MHz)**	**Clock cycles (million)**	**Block size (bits)**	**Area (slices)**	**Number of RAM Blocks**	**T/put (Kbps)**	**T/put / area (Kbps/slice)**
xc3s50-5	73.3	55.5	1024	224	4	1.4	0.006
xc5vlx30-3	115.3	55.5	1024	101	3	2.1	0.02
Spartan 3A-5 [56]	102	51	1024	302	3	2	0.007
Virtex 6-3 [56]	278	51.5	1024	145	1	5.4	0.04

CHAPTER 5

CRYPTOGRAPHIC PROCESSOR INTEGRATION

This chapter summarizes the integration of the main controller with the cryptographic coprocessors. The area figures of the resultant cryptographic processor for different configurations are presented together with figures from similar works in the literature. It is then followed by the throughput performance figures of the present work and other embedded implementations. Finally, the coprocessor interface is explained in detailed by means of an AES program example and corresponding simulation results.

5.1 Integration

The cryptographic coprocessors are enclosed inside wrappers, which make them behave like regular RAMs, and combined together with the data RAM of the main controller to form the memory block. I/O ports of each coprocessor are entries in the memory address map of the main controller, and can be accessed by the software via LOAD/STORE instructions. The finalized design is shown in Figure 5.1.

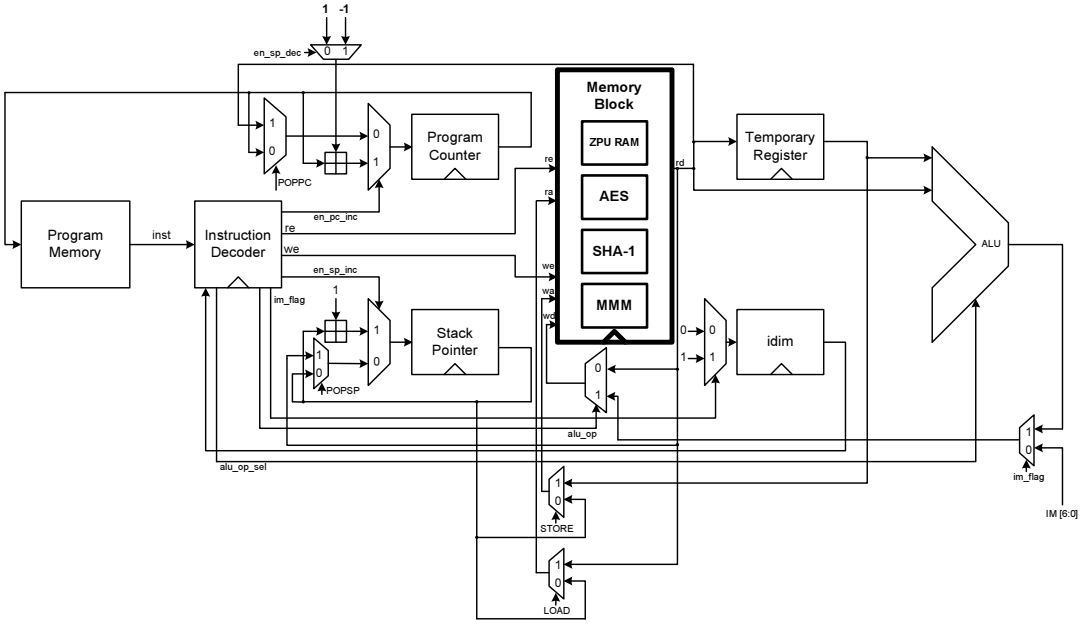

Figure 5.1 ZPU block diagram with integrated coprocessors.

5.2 Implementation Results

Table 5.1 summarizes the implementation results for the cryptographic processor with integrated coprocessors for both 2KB and 8KB program memory options. The slice count is 1982 and the maximum achievable frequency is 63.4 MHz on the smallest Virtex-5 device. These results are compared with two reference designs. However, it should be noted that we have not been able to find any other processor, which integrates all three AES, SHA-1 and MMM coprocessors. The closest implementation is the processor presented in [57] with built-in AES and SHA-1 functionality. In addition, we also take the implementation results of a RSA-specific processor presented in [58].

Table 5.1 Cryptographic processor implementation results.

Device / Implementation	Freq (MHz)	Area (slices)	Number of RAM Blocks
Xilinx Virtex-5 (xc5vlx30-3) With coprocessors (8K program memory)	63.4	1982	4
Xilinx Virtex-5 (xc5vlx30-3) With coprocessors (2K program memory)	63.4	1982	0
Xilinx Virtex (xcv1000e) (AES and SHA-1 crypto processor) [57]	24.2	7247	20
Xilinx (xcv1000-6) (RSA crypto processor) [58]	30	936	-

5.3 Performance Results

In Table 5.2, the estimated performance figures of the cryptographic processor for AES, SHA-1 and MMM operations are presented. In the estimation, the standalone performance of each coprocessor is multiplied by a factor in order to take the software overhead into account. It is hard to come up with a definite factor for the AES and SHA-1 operations, since the software overhead depends on the specific IPSec protocol being implemented as well as the actual IPSec packet size. However, the simulation results from sample implementations reveal a factor of 0.2 and 0.25 for the AES and SHA-1 operations. In the case of RSA, software overhead is almost negligible as even the modular multiplication operation itself takes thousand of cycles. Even the factor of 0.9 applied in the calculations is too conservative.

The performance of the cryptographic processor is compared against software-only embedded implementations presented in [59-61] in order to demonstrate the hybrid approach applied in this study.

Table 5.2. Throughput performances.

Implementation	Throughput
ZPU's AES coprocessor, performance @ 63.4 MHz Virtex-5 xc5vlx30-3 (divided by 5, for software overhead)	115.9 Mbps
ZPU's SHA-1 coprocessor, performance @ 63.4 MHz Virtex-5 xc5vlx30-3 (divided by 4, for software overhead)	101.4 Mbps
ZPU's MMM coprocessor, performance @ 63.4 MHz Virtex-5 xc5vlx30-3 (divided by 1.25, for software overhead)	0.92 Kbps
AES performance on ARM9 processor @ 200 MHz [59]	30.8 Mbps
SHA-1 performance on ARM9 processor @ 200 MHz [59]	64 Mbps
RSA performance on StrongARM processor @ 200 MHz [60]	0.53 Kbps
AES performance on ARM processor @ 200 MHz [61]	47 Mbps

5.4 Coprocessor Interface

The implemented coprocessor plug-in interface is simple in its nature. In each RAM-like coprocessor, one of the 16-bit memory addresses (the address 0x-F--) act as the virtual command-status register (CSR) to provide an interface between the main processor and the coprocessors.

First, the input memories of the chosen coprocessor are filled by the microprocessor. Then a write is issued to the corresponding CSR, instructing the coprocessor to start its operation. This raises the coprocessor active flag. The rising edge of the active flag is converted to a "start" pulse for the coprocessor, while the active flag itself acts as a system-wide busy signal which halts the ZPU core. Once the coprocessor is done, it generates a "ready" pulse which pulls down the active signal. The system-wide busy is also pulled down, allowing the processor to continue its program execution with the next instruction in line. This CSR scheme is explained below by means of a simple AES example.

We start with the C program segment given below. In this program segment, first, 4 words (128 bits) of data from the message address space are copied into the AES input registers. Then the AES CSR register is written. The data written into this address is unimportant as it is a virtual register. The write operation into this register triggers the AES start pulse, and the AES coprocessor starts its operation. During its operation, the AES busy signal is active and the main

program execution is halted. This is not shown in the C code. As soon the AES completes its operation, the ready pulse is generated, which deactivates the busy signal. Program execution continues with the copying of the AES results from the AES output registers onto the original input message.

```
0023
0024   AES_in[0] =  MSG[0];
0025   AES_in[1] =  MSG[1];
0026   AES_in[2] =  MSG[2];
0027   AES_in[3] =  MSG[3];
0028
0029   *AES_CSR = 1;
0030
0031   MSG[0] = AES_out[0];
0032   MSG[1] = AES_out[1];
0033   MSG[2] = AES_out[2];
0034   MSG[3] = AES_out[3];
0035
```

The corresponding assembler code segment is shown below. It should be noted how a simple assignment as in line 0025 in the C code is replaced by 6 lines of code (0216 to 0221) in the assembler code. Furthermore, due to the 7-bit maximum immediate value limit of the ZPU architecture, larger numbers can be entered with two consecutive IM instructions as seen throughout the assembler code. This results not only in the loss of program memory space but execution time as well.

```
0208
0209   im 10   // im 10 followed by im 0
0210   im 0    // is equivalent to im 1280 = im 0x0500
0211   loadsp 0
0212   load
0213   im 32   // im 32 followed by im 0
0214   im 0    // is equivalent to im 4096 = im 0x1000
0215   store   // mem[0x1000] = mem[0x0500] -> AES_in[0] = MSG[0];
0216   im 10   // im 10 followed by im 4
0217   im 4    // is equivalent to im 1284 = im 0x0504
0218   load
0219   im 32   // im 32 followed by im 4
0220   im 4    // is equivalent to im 4100 = im 0x1004
0221   store   // mem[0x1004] = mem[0x0504] -> AES_in[1] = MSG[1];
0222   im 10   // im 10 followed by im 8
0223   im 8    // is equivalent to im 1288 = im 0x0508
0224   load
0225   im 32   // im 32 followed by im 8
0226   im 8    // is equivalent to im 4104 = im 0x1008
0227   store   // mem[0x1008] = mem[0x0508] -> AES_in[2] = MSG[2];
0228   im 10   // im 10 followed by im 12
0229   im 12   // is equivalent to im 1292 = im 0x050C
0230   load
```

```
0231   im 32   // im 32 followed by im 12
0232   im 8    // is equivalent to im 4108 = im 0x100C
0233   store   // mem[0x100C] = mem[0x050C] -> AES_in[3] = MSG[3];
0234   storesp 4
0235   im 1
0236   nop
0237   im 62   // im 62 followed by im 0
0238   im 0    // is equivalent to im 7936 = im 0x1F00
0239   store   // mem[0x1F00] = 1 -> *AES_CSR = 1;  // ZPU halts
0240   im 34   // im 34 followed by im 0
0241   im 0    // is equivalent to im 4352 = im 0x1100
0242   load
0243   loadsp 4
0244   store   // mem[0x0500]=mem[0x1100] -> MSG[0] = AES_out[0];
0245   im 34   // im 34 followed by im 4
0246   im 4    // is equivalent to im 4356 = im 0x1104
0247   load
0248   im 10   // im 10 followed by im 4
0249   im 4    // is equivalent to im 1284 = im 0x0504
0250   store   // mem[0x0504]=mem[0x1104] -> MSG[1] = AES_out[1];
0251   im 34   // im 34 followed by im 8
0252   im 8    // is equivalent to im 4360 = im 0x1108
0253   load
0254   im 10   // im 10 followed by im 8
0255   im 8    // is equivalent to im 1288 = im 0x0508
0256   store   // mem[0x0508]=mem[0x1108] -> MSG[2] = AES_out[2];
0257   im 34   // im 34 followed by im 12
0258   im 12   // is equivalent to im 4364 = im 0x110C
0259   load
0260   im 10   // im 10 followed by im 12
0261   im 12   // is equivalent to im 1292 = im 0x050C
0262   store   // mem[0x050C]=mem[0x110C] -> MSG[3] = AES_out[3];
```

Finally, simulation results for the execution of the assembler code from lines 0235 to 0242 is shown in Figure 5.2. This particular segment is chosen for various reasons. First of all, it is the code segment, where the AES coprocessor is instantiated. As a result, the program execution halts until the completion of the AES operation.

Furthermore, this code segment uses the most commonly used instructions, namely IM, LOAD and STORE, in the realization of IPSec protocols on the resultant processor. The entry of a large immediate number into the stack via two consecutive IM instructions and the behavior of the IDIM flag are also demonstrated in the simulation waveforms of the chosen segment.

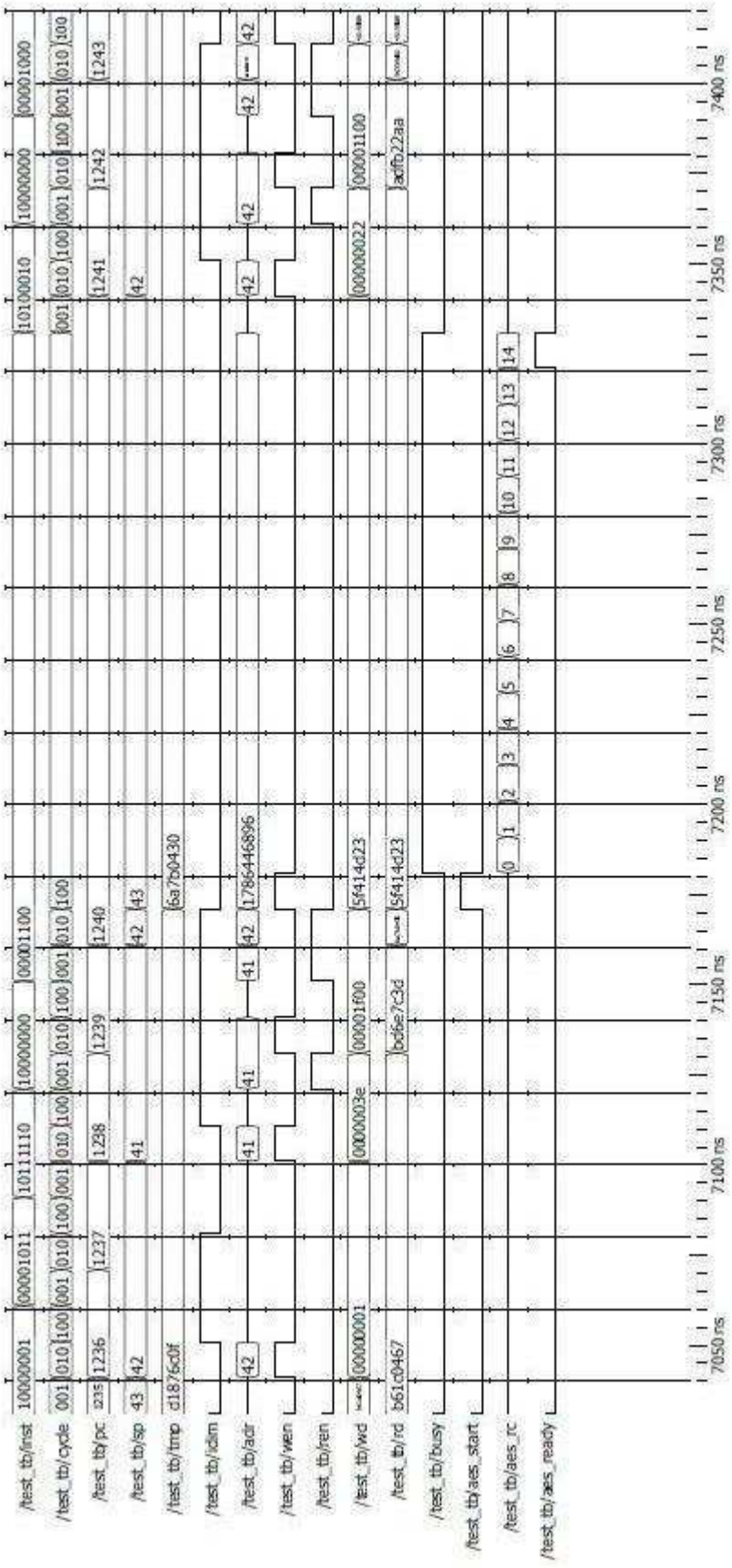

Figure 5.2 Simulation results for the execution of the assembler code.

CHAPTER 6

IPSEC PROTOCOL IMPLEMENTATION EXAMPLES

In this chapter, three software examples are given in order to present the operation and use of the cryptographic processor in the implementation of IPSec protocols and components. Each example demonstrates the use of one of the coprocessors. The first example uses the AES coprocessor in order to implement the Counter with Cipher Block Chaining-Message Authentication (CCM) mode, which is an optional combined encryption and authentication scheme of the IPSec protocol suite. It is followed by the Hash based Message Authentication Code (HMAC), which utilizes the SHA-1 coprocessor. The last example uses the Montgomery modular multiplier coprocessor in order to implement the RSA encryption/decryption algorithm, which is an important component of the Internet Key Exchange (IKE) protocol of IPSec.

All three codes are written as functions, which can be called via a “main” function running on the cryptographic processor. It is the duty of the main function to manipulate the IP packet and call the appropriate functions. The structure of the main function and how it communicates with the external world is explained in the next section, which is followed by individual sections for each software example.

6.1 IP Packet Handling

In our work, the IP packet is assumed to be received in classified form, which means that the input data is sent to the corresponding RAM addresses in ZPU. Then, we process the payload according to this information. A simple flow of this algorithm and working principle of ZPU can be shown as follows:

```
void main ( ) ;
label: *ZPU_CSR=0 ; /* ZPU command status register is set to 0
                       initially. */
          while (*ZPU_CSR = 0)
         /* Exterior controller writes data into RAM by pulling
            ext_sel to 1. It also sets ZPU_CSR to 1, signaling
            that data is ready. */
```

```
            .
            . => IP packet handling part is done.
            .
            .
            . => IP data processing is done (CCM, HMAC, RSA, etc.).
            .

          *ZPU_RDY = 1 ;  /* ZPU ready register is set to 1 when
                             process is  finished, for signaling to
                             the exterior controller that ZPU is
                             done. */
           /* Exterior controller reads. */
go to label
```

First, ZPU exits from the reset condition and waits for the ZPU_CSR to be set to 1 in an initial while loop. Then, the external controller sets ext_sel to 1, which enables the sending of the message and CSR data to the corresponding places in RAM and writes them in. While doing this, ZPU is halted until the process is finished. When ext_sel is set to 0, ZPU continues to its while loop. Meantime, it will detect that ZPU_CSR is set to 1 and break the loop to perform its operations. It will decide the appropriate operation according to the written configuration and message data. Then, it will set ZPU_RDY to 1 every time its work is finished. Actually, that ZPU_RDY address is not a physical RAM space, it is just a 1-bit status register which is set to 1 when ZPU writes and set to 0 when ext_sel is activated.

After all these, ZPU will read the data whenever it needs. By the way, ZPU returns to the beginning, which is named as “label” in the pseudo-code. Then, it will wait for 1 to be written in ZPU_CSR.

6.2 Counter with Cipher Block Chaining–Message Authentication Code (CCM)

Counter with Cipher Block Chaining-Message Authentication Code (CCM) [39] is used to provide assurance of the privacy and the authenticity of data by combining the techniques of the Counter (CTR) mode [62] and the Cipher Block Chaining-Message Authentication Code (CBC-MAC) algorithm [63]. CCM is based on an approved symmetric key block cipher algorithm whose block size is 128 bits, such as the Advanced Encryption Standard (AES) algorithm which is explained in the previous chapter.

CCM can be considered as a mode of operation of the block cipher algorithm. A single key to the block cipher must be established beforehand among the parties to the data. So, CCM should be

implemented within a well-designed key management structure. The security properties of CCM depend on the secrecy of this key.

CCM is intended for use in a packet environment. All of the data should be available in storage before CCM is applied. CCM is not designed to support partial processing or stream processing. Three inputs to CCM are:

- data that will be both authenticated and encrypted which is called the payload,
- associated data that will be authenticated but not encrypted,
- a unique value called nonce, which is assigned to the payload and the associated data.

CCM consists of two related processes: generation-encryption and decryption-verification. Only the forward cipher function of the block cipher algorithm is used within these primitives. In generation-encryption, cipher block chaining is applied to the payload, the associated data, and the nonce to generate a message authentication code (MAC). Then, counter mode encryption is applied to the MAC and the payload, to transform them into an unreadable form which is called the cipher text. Therefore, it can be seen that CCM generation-encryption expands the size of the payload by the size of the MAC. In decryption-verification, counter mode decryption is applied to the supposed cipher text to recover the MAC and the corresponding payload. Then, cipher block chaining is applied to the payload, which is the received data, and the received nonce to verify the correctness of the MAC. A successful verification provides assurance that the payload and the associated data originated from a source with access to the key.

A MAC provides stronger assurance of authenticity than a checksum or an error detecting code. The verification of a checksum or an error detecting code is designed to detect only accidental modifications of the data, while the verification of a MAC is designed to detect intentional, unauthorized modifications of the data, as well as accidental modifications.

6.2.1 Description of CCM

As mentioned before, two CCM processes are called generation-encryption and decryption-verification. The order of the steps of these two processes is a little bit flexible. For example, the generation of the counter blocks may occur at any time before they are used. In fact, the counter blocks may be generated in advance to be considered as inputs to the processes.

The below algorithm explains the generation-encryption process. The input data to the generation-encryption process is a valid nonce, a valid payload string and a valid associated data string, which are formatted according to the formatting function. The CBC-MAC mechanism is applied to the formatted data to generate a MAC, whose length is a prerequisite. Counter mode

encryption, which requires a sufficiently long sequence of counter blocks as input, is applied to the payload string and separately to the MAC. The resulting data which is called the ciphertext (denoted *C*) is the output of the generation-encryption process.

Prerequisites:
block cipher algorithm,
key *K*,
counter generation function,
formatting function,
MAC length *Tlen*,

Inputs:
valid nonce *N* (salt + initialization vector (IV)),
valid payload *P* of length *Plen* bits (= *M* blocks – each block is 128 bits long),
valid associated data *A* of length *Alen* bits (= *D* blocks – each block is 128 bits long),

Outputs:
ciphertext *C*.

Steps:
1. Apply the formatting function to (*M*, *A*, *P*) to produce *D*+*M* blocks $B_1, B_2, \ldots, B_{D+M}$. IV is added at the beginning of the block as $B_0 = IV$.
2. Set $X_1 = E_K(B_0)$.
3. For *i* = 1 to *D*+*M*, do $X_{i+1} = E_K(X_i \oplus B_i)$.
4. Set $T = \mathrm{MSB}_{Tlen}(X_{D+M+1})$.
5. Apply the counter generation function to generate the counter blocks $Ctr_0, Ctr_1, \ldots, Ctr_M$, where $M = \lceil Plen / 128 \rceil$.
6. For *j*=0 to *M*, do $S_j = E_K(Ctr_j)$.
7. Set $S = S_1 \| S_2 \| \ldots \| S_M$.
8. Return $C = (P \oplus \mathrm{MSB}_{Plen}(S)) \| (T \oplus \mathrm{MSB}_{Tlen}(S_0))$.

The input to the decryption-verification process, which is described in the below pseudo-code, is a supposed ciphertext, an associated data string and the nonce that is believed to be used in the

generation of the supposed ciphertext. Counter mode decryption is applied to the supposed ciphertext to produce the corresponding MAC and payload. If the nonce, the associated data string and the payload are valid, then these strings are formatted into blocks according to the formatting function and the CBC-MAC mechanism is applied to verify the MAC. If the verification succeeds, then the decryption-verification process returns the payload as output. Otherwise, only the error message INVALID is returned.

When the error message INVALID is returned, the payload *P* and the MAC *T* should not be displayed. Moreover, the implementation should ensure that an unauthorized party cannot distinguish if the error message results from Step 7 or from Step 10, for example from the timing of the error message.

Prerequisites:
block cipher algorithm,
key *K*,
counter generation function,
formatting function,
valid MAC length *Tlen*,

Inputs:
nonce *N* (salt + initialization vector (IV)),
associated data *A* of length *Alen* bits (= *D* blocks – each block is 128 bits long),
supposed ciphertext *C* of length *Clen* bits (= *R* blocks – each block is 128 bits long),

Output:
either the payload *P* of length *Plen* bits (= *M* blocks – each block is 128 bits long) or INVALID.

Steps:
1. If $Clen \leq Tlen$, then return INVALID.
2. Apply the counter generation function to generate the counter blocks $Ctr_0, Ctr_1, ..., Ctr_M$, where $M = \lceil (Clen - Tlen)/128 \rceil$.
3. For j=0 to M, do $S_j = E_K(Ctr_j)$.
4. Set $S = S_1 \| S_2 \| ... \| S_M$.
5. Set $P = \mathrm{MSB}_{Clen-Tlen}(C) \oplus \mathrm{MSB}_{Clen-Tlen}(S)$.

6. Set $T = \text{MSB}_{Tlen}(C) \oplus \text{MSB}_{Tlen}(S_0)$.

7. If *N*, *A* or *P* is not valid, then return INVALID. Else, apply the formatting function to (*N*, *A*, *P*) to produce the blocks $B_1, B_2, \ldots, B_{D+M}$.

8. Set $X_1 = E_K(B_0)$.

9. For *i*=1 to *D*+*M*, do $X_{i+1} = E_K(X_i \oplus B_i)$.

10. If $T \neq \text{MSB}_{Tlen}(X_{D+M+1})$, then return INVALID. Else, return *P*.

6.2.2 Software Overview of AES-CCM

AES-CCM is performed on 16-byte (128-bit) blocks. However, since the processor data bus is 32-bits wide, AES input is not directly sent to the core in 128-bit format. At first, necessary data (such as flags, nonce, payload, AAD) is read from corresponding RAM addresses and then the 128-bit input to the AES core is formed and stored in four 32-bit temporary variables in the software. These temporary variables are mapped to 32-bit wide RAM locations (words) in the actual hardware. 4-word AES input read from the temporary memory locations is sent to the four AES input registers to be processed. The AES input registers are also mapped to specific locations in the processor's address space. Once the AES inputs are transferred, a write is issues to the virtual AES command/status register signaling the AES to start its encryption. This sets the physical "busy" signal in the processor and halts its program execution, until the AES coprocessor completes its run and clears the "busy" signal by issuing the "encrypted block ready" signal. The encrypted block is then read from four (4x32=128-bit) AES output addresses. AES is performed for blocks of cipher block chaining and counter mode parts. In the end, the cipher text is formed from the encrypted counter blocks and MAC (MAC length is given as input - *Tlen*).

For IPSec purposes, certain inputs are fixed in the standard, such as the length of AAD and the length of nonce. For example, AAD length is stated as 8 or 12 bytes in the standard [64]. Therefore, simplifications can be done on the software code to cover only these IPSec properties. The algorithm is rewritten according to this, as follows:

$B_0 = \{ flags, nonce, lm \}$

$X_1 = E_K(B_0)$

$B_0 = \{ la\ (2\,bytes), AAD, zeros\ (to\ fit 16\,bytes) \}$

$X_2 = E_K(X_1 + B_1)$

for i = 2 to N,

$$B_i = \{payload(, zeros\ to\ fit\ 16\ bytes\ if\ necessary)\}$$

$$X_{i+1} = E_K(X_i \oplus B_i)\ .$$

$$A_{i-1} = \{flags, nonce, cnt_{i-1}\}$$

$$S_{i-1} = E_K(A_{i-1})$$

$$C_{i-2} = B_i \oplus S_{i-1}$$

end for

$$T = \mathrm{MSB}_{Tlen}(X_{N+1})$$

$$A_0 = \{flags, nonce, cnt_0\}$$

$$S_0 = E_K(A_0)$$

$$U = T \oplus S_0$$

$$C = C_0 \| C_1 \| ... \| C_{N-2} \| U$$

In Figure 6.1, the block diagram of this operation can be seen in detail.

The flowchart of the operation is given in Figure 6.2.

The decryption-verification is the reverse process of the above operation, as mentioned before. Therefore, the block diagram and the flowchart will be similar to encryption with small modifications for decryption.

6.3 Hash-based Message Authentication Code (HMAC)

In communications, providing a way to check the integrity of information transmitted over or stored in an unknown medium is a major necessity. Mechanisms that provide such integrity checks based on a secret key are called message authentication codes (MACs), as mentioned in previous section. A MAC that uses an approved cryptographic hash function in conjunction with a secret key is called hash-based message authentication code (HMAC) [65,59].

The main goals behind the HMAC construction [66] are:

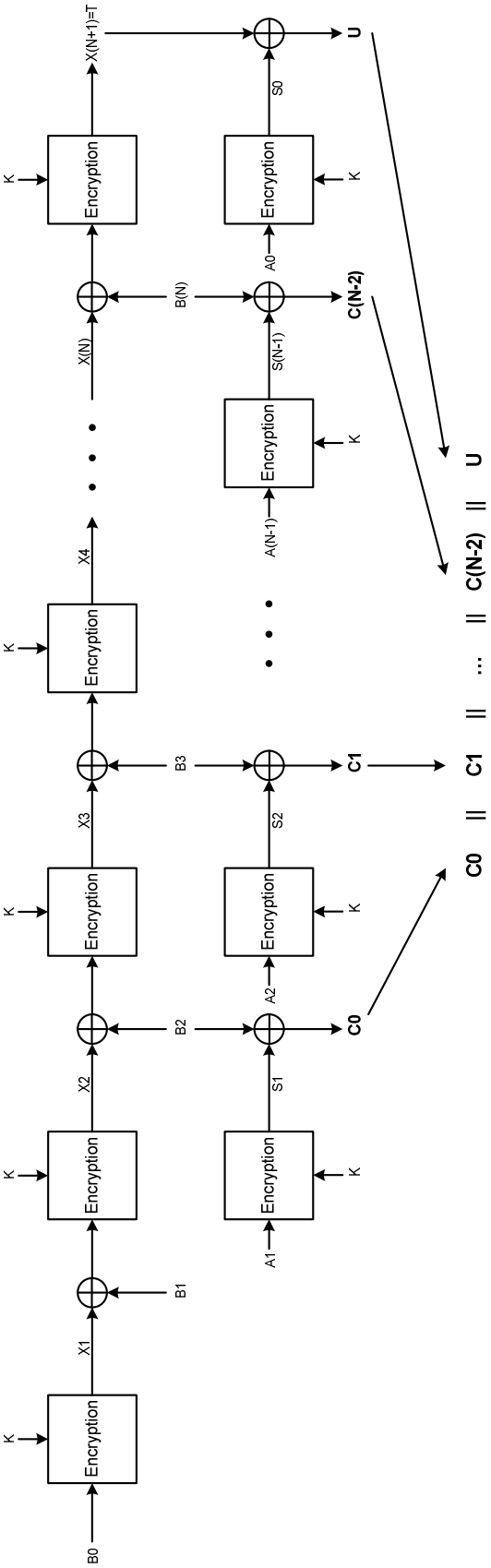

Figure 6.1 AES-CCM block diagram.

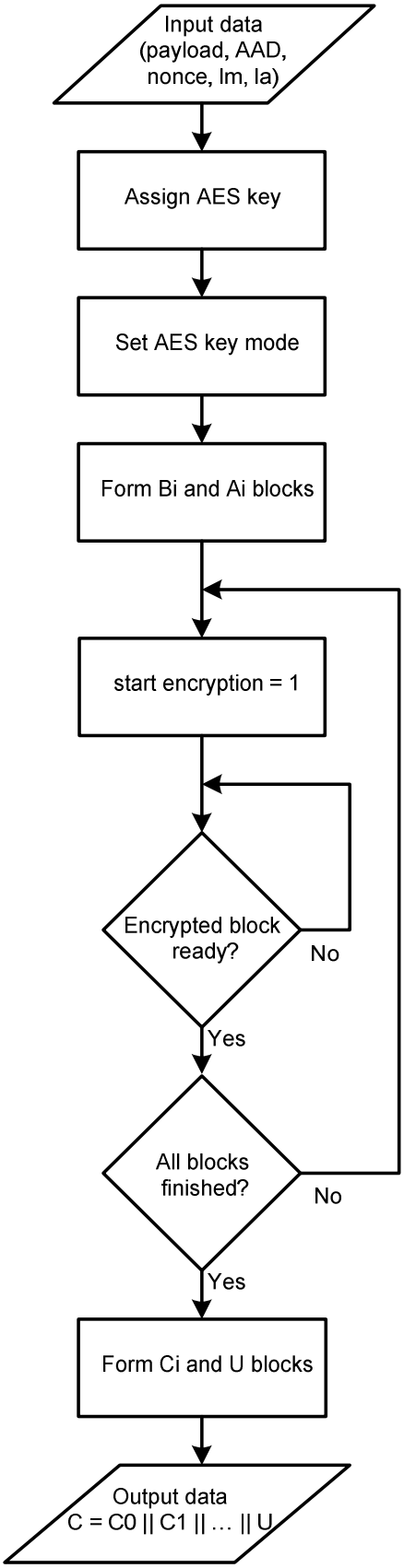

Figure 6.2 Flowchart of AES-CCM.

- to use available hash functions without modifications,
- to preserve the original performance of the hash function,
- to use and handle keys in a simple way,
- to have a good cryptographic analysis of the strength of the authentication mechanism on the underlying hash function,
- to allow easy replaceability of the underlying hash function, in case that faster or more secure hash functions are available in the future.

Any iterative cryptographic hash function, such as SHA-1, SHA-224 … etc., may be used in the calculation of an HMAC. So, the resulting MAC algorithm is termed as HMAC-SHA-1, HMAC-SHA-224 … etc., accordingly. The size of the output of HMAC is the same as that of the underlying hash function (160, 256 or 512 bits in case of SHA-1, SHA-256 and SHA-512, respectively), although it can be truncated if desired.

6.3.1 Description of HMAC

In the definition of HMAC, the cryptographic hash function is denoted by *H* and the secret key is denoted by *K*. The byte-length of blocks on which *H* operates iteratively, is denoted by *B* ($B = 64$ for SHA-1, SHA-224, SHA-256 and $B = 128$ for SHA-384, SHA-512). The byte-length of hash function outputs is denoted by *L* (L=20, 28, 32, 48 and 64 for SHA-1, SHA-224, SHA-256, SHA-384 and SHA-512, respectively). The authentication key *K* can be of any length up to *B*, the block length of the hash function.

Two fixed and different strings *ipad* and *opad* are defined as follows (the '*i*' and '*o*' are mnemonics for inner and outer):

ipad = the byte `0x36` repeated *B* times,
opad = the byte `0x5C` repeated *B* times.

To compute a MAC over the data '*text*' using the HMAC function, the following operation is performed:

$$MAC(text)_t = HMAC(K, text)_t = H((K_0 \oplus opad) \,||\, H((K_0 \oplus ipad) \,||\, text))_t \quad (6.1)$$

Step by step process of the HMAC algorithm is explained as follows:

Steps:

1. If the length of $K = B$: set $K_0 = K$. Go to step 4.
2. If the length of $K > B$: hash K to obtain an L byte string, then append (B-L) zeros to create a B-byte string K_0 (i.e., $K_0 = H(K) \| 00...00$). Go to step 4.
3. If the length of $K < B$: append zeros to the end of K to create a B-byte string K_0 (i.e., if K is 20 bytes in length and $B = 64$, then K will be appended with 44 zero bytes $0x00$).
4. XOR K_0 with $ipad$ to produce a B-byte string: $K_0 \oplus ipad$.
5. Append the stream of data '$text$' to the string resulting from step 4: $(K_0 \oplus ipad) \| text)$.
6. Apply H to the stream generated in step 5: $H((K_0 \oplus ipad) \| text)$.
7. XOR K_0 with $opad$: $K_0 \oplus opad$.
8. Append the result from step 6 to step 7: $(K_0 \oplus opad) \| H((K_0 \oplus ipad) \| text)$.
9. Apply H to the result from step 8: $H((K_0 \oplus opad) \| H((K_0 \oplus ipad) \| text))$.
10. Select the leftmost t bytes of the result of step 9 as the MAC.

6.3.2 Software Overview of HMAC-SHA-1-96

HMAC-SHA-1-96 is performed on 64-byte (512-bit) blocks. However, a SHA-1 input is not directly sent to the core in 512-bit format. At first, necessary data (such as key and payload) is read from corresponding RAM addresses and then the 512-bit input to the SHA-1 core is formed and stored in sixteen 32-bit temporary variables. As in the case of AES, once the SHA-1 coprocessor inputs are ready, a write is issued to the virtual SHA-1 command/status register and the coprocessor starts its operations halting the main processor's program execution via the "busy" signal. After SHA-1 completes processing the 512-bit input block, it releases the busy and the processor continues program execution. At this step, the "hashed data" can either be read from the SHA-1 output registers, or hashing can be continued with new inputs.

For IPSec, certain inputs are fixed in the standard. For example, the length of payload is fixed to multiples of 512 bits and the key length is fixed to 20 bytes (160 bits – 5 RAM locations) [67]. Therefore, simplifications can be done on the software code to cover only these IPSec properties. The algorithm is rewritten according to this, as follows:

$B_0[0:4] = (K \oplus ipad); B_0[5:15] = ipad\ \ bytes$

$B_1 = text[0:15]$

$B_2 = text[16:31]$

.

.

.

$B_N = text[16\text{x}(N-1):(16\text{x}(N-1))+15]$

$clear = 1;$

$h[0:4] = IV;$

$SHA\_in = B_0;$

$start = 1;$

$for(i = 1\ to\ \ N)$

$h[0:4] = SHA\_out;$

$SHA\_in = B_i;$

$start = 1;$

$end\ \ for$

$c = SHA\_out;$

$A_0[0:4] = (K \oplus opad); A_0[5:15] = opad\ \ bytes$

$A_1[0:4] = c; A_1[5:15] = SHA\_padding$

$clear = 1;$

$h[0:4] = IV;$

$SHA\_in = A_0;$

$start = 1;$

$h[0:4] = SHA\_out;$

$SHA\_in = A_1;$

$start = 1;$

$Y = SHA\_out[0:2]$

In the algorithm pseudo-code given above, the “clear” signal is given in addition to the “start” in order to identify the first 512-bit block input of the hashing operation. The purpose of this identification is to select initialization vector as *h[0:4]*. Unlike the “start” signal, it does not

activate the busy and halt program execution. The flowchart of the operation is given in Figure 6.3.

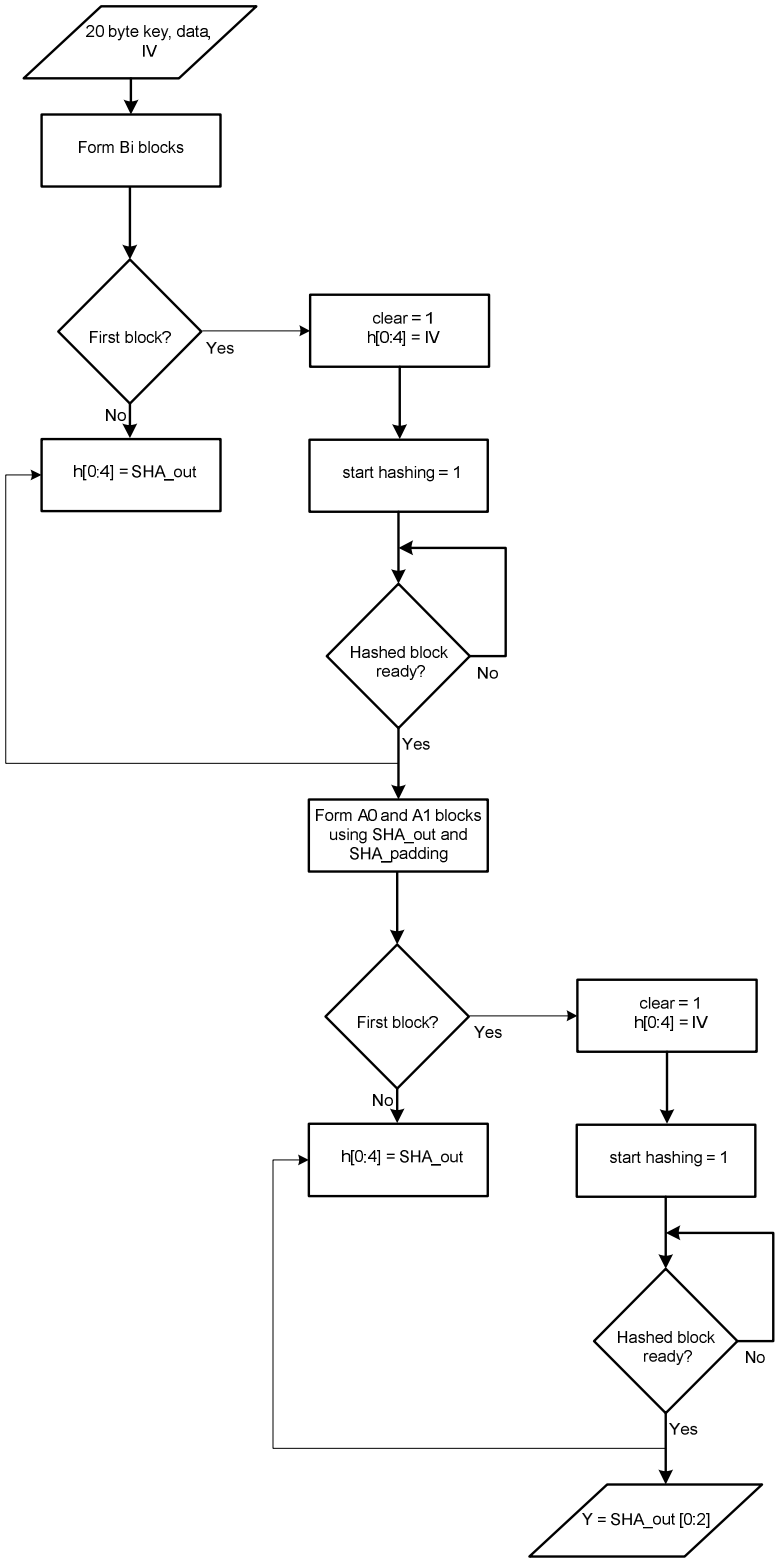

Figure 6.3 Flowchart of HMAC-SHA-1-96.

6.4 RSA Encryption and Decryption for Internet Key Exchange

The public key algorithm RSA (which stands for Ron Rivest, Adi Shamir and Leonard Adleman, who first publicly described it at MIT, in 1977) is the first algorithm known to be suitable for both asymmetric encryption/decryption and signature generation/verification purposes, and it was one of the first great advances in public key cryptography. RSA, which became patent free and released to the public domain in 2000, is the most widely used public key algorithm in electronic commerce protocols and believed to be secure given sufficiently long keys and the use of up-to-date implementations.

RSA involves a public key and a private key. The public key can be known to everyone and it is used for encrypting messages. Messages encrypted with the public key can only be decrypted using the private key.

RSA gets its security from integer factorization problem. Difficulty of factoring large numbers is the basis of security of RSA (512, 1024 or 2048 bits long, generally).

RSA is much slower than symmetric cryptosystems. In practice, one typically encrypts a secret message with a symmetric algorithm, encrypts the (comparatively short) symmetric key with RSA and transmits both the RSA-encrypted symmetric key and the symmetrically-encrypted message to the recipient. This procedure raises additional security issues. For instance, it is of ultimate importance to use a strong random number generator for the symmetric key. Otherwise, an eavesdropper could bypass RSA by guessing the symmetric key.

RSA algorithm is one of the simplest public-key cryptosystems in terms of mathematical complexity, which makes it relatively easy to understand. However, the same can not be said for the hardware implementation.

6.4.1 Description of RSA

As mentioned in the previous chapter, RSA encryption and decryption operations are defined as:

$$c = m^e \bmod n \tag{6.2}$$

and

$$m = c^d \bmod n \tag{6.3}$$

(m: N-bit message, c: N-bit ciphertext, n: N-bit RSA modulus - a product of 2 distinct odd primes, e: E-bit RSA public key - $E << N$ - , d: N-bit private key).

RSA is based on modular exponentiation operation, which is in turn based on modular multiplication. Assume that; $x = x_{N-1}x_{N-2}\ldots x_0$, $y = y_{E-1}y_{E-2}\ldots y_0$, $z = z_{N-1}z_{N-2}\ldots z_0$ and $w = w_{N-1}w_{N-2}\ldots w_0$. Then, $w = x^y \bmod z$ can be calculated using the algorithm below:

$$
\begin{aligned}
&M_0 = x,\ R_0 = 1 \\
&\text{for } i = 0 \text{ to } E-1 \\
&\quad R_{i+1} = \begin{cases} R_i \times M_i \bmod z & \text{if } y_i = 1 \\ R_i & \text{else } (y_i = 0) \end{cases} \\
&\quad M_{i+1} = M_i \times M_i \bmod z \\
&\text{end for} \\
&w = R_E
\end{aligned}
\tag{6.4}
$$

Modular exponentiation can easily be implemented if an ideal modular multiplication operator exists. As explained before, it is costly to implement an ideal modular multiplication module in hardware and Montgomery multiplication algorithm [33,36] is preferred for hardware implementation, instead.

The undesired 2^{-N} factor (mentioned in Chapter 4, Section 3) that Montgomery modular multiplication introduces has to be taken care of by modifying the original modular exponentiation algorithm, which leads to the Montgomery modular exponentiation. Then, the Montgomery modular exponentiation algorithm is re-defined as:

$$
\begin{aligned}
&k \equiv 2^{2N} \bmod z \\
&M_0 = \mathrm{MMM}(x, k, z) \\
&R_0 = \mathrm{MMM}(1, k, z) \\
&\text{for } i = 0 \text{ to } E-1 \\
&\quad R_{i+1} = \begin{cases} \mathrm{MMM}(R_i, M_i, z) & \text{if } y_i = 1 \\ R_i & \text{else} \end{cases} \\
&\quad M_{i+1} = \mathrm{MMM}(M_i, M_i, z) \\
&\text{end for} \\
&w = \mathrm{MMM}(R_E, 1, z)
\end{aligned}
\tag{6.5}
$$

where

$$w = x^y \bmod z\,. \tag{6.6}$$

6.4.2 Software Overview of RSA

RSA encryption algorithm has different modes, such as RSA-512, RSA-1024, RSA-2048, depending on the length of the inputs. Therefore, RSA is performed on 64, 128 or 256 byte (512, 1024 or 2048 bits) blocks. However, an RSA input is not directly sent to the MMM coprocessor in 512, 1024 or 2048 bits format. At first, necessary data (such as message, keys, modulus and K constant) is read from corresponding RAM addresses. As memory locations are 32-bit wide, MMM inputs are sent to corresponding 16(x32=512-bit), 32(x32=1024-bit) or 64(x32=2048-bit) input addresses to be processed. The rest is similar to AES and SHA-1 coprocessor operation. A write into the virtual MMM command/status register starts its operation, activates the busy signal, and halts the processor program execution. When MMM output is ready, busy signal is released and processor program continues its run by transferring the multiplication result from the MMM output registers to target addresses inside the memory.

RSA algorithm can directly be implemented in software. Recall the algorithm in original format:

$$c = m^e \bmod n$$

$$\begin{aligned}
&K \equiv 2^{2N} \bmod n \\
&M_0 = \mathrm{MMM}(m, K, n) \\
&R_0 = \mathrm{MMM}(1, K, n) \\
&\text{for } i = 0 \text{ to } E-1 \\
&\quad R_{i+1} = \begin{cases} \mathrm{MMM}(R_i, M_i, n) & \text{if } e_i = 1 \\ R_i & \text{else} \end{cases} \\
&\quad M_{i+1} = \mathrm{MMM}(M_i, M_i, n) \\
&\text{end for} \\
&c = \mathrm{MMM}(R_E, 1, n)
\end{aligned} \tag{6.7}$$

Then, the algorithm can be rewritten for software implementation as follows:

```
// m = MMM(m, K, n) //
//
mem(MMM_A) ← mem(m);
mem(MMM_B) ← mem(K);
mem(MMM_C) ← mem(n);
start = 1;
mem(m) ← mem(MMM_Y);
```

```
// r = MMM(1, K, n) //
//
mem(MMM_A) ← mem(1);
mem(MMM_B) ← mem(K);
start = 1;
mem(r) ← mem(MMM_Y);
for i=0 to E-1
    mem(MMM_B) ← mem(m);
    if e(i)
        // r = MMM(r, m, n) //
        //
        mem(MMM_A) ← mem(r);
        start = 1;
        mem(r) ← mem(MMM_Y);
    end if
    // m = MMM(m, m, n) //
    //
    mem(MMM_A) ← mem(m);
    start = 1;
    mem(m) ← mem(MMM_Y);
end for
  // r = MMM(r, 1, n) //
  //
  mem(MMM_A) ← mem(r);
  mem(MMM_B) ← mem(1);
  start = 1;
  mem(r) ← mem(MMM_Y);
c ← mem(r);
```

The flowchart of RSA encryption is given in Figure 6.4.

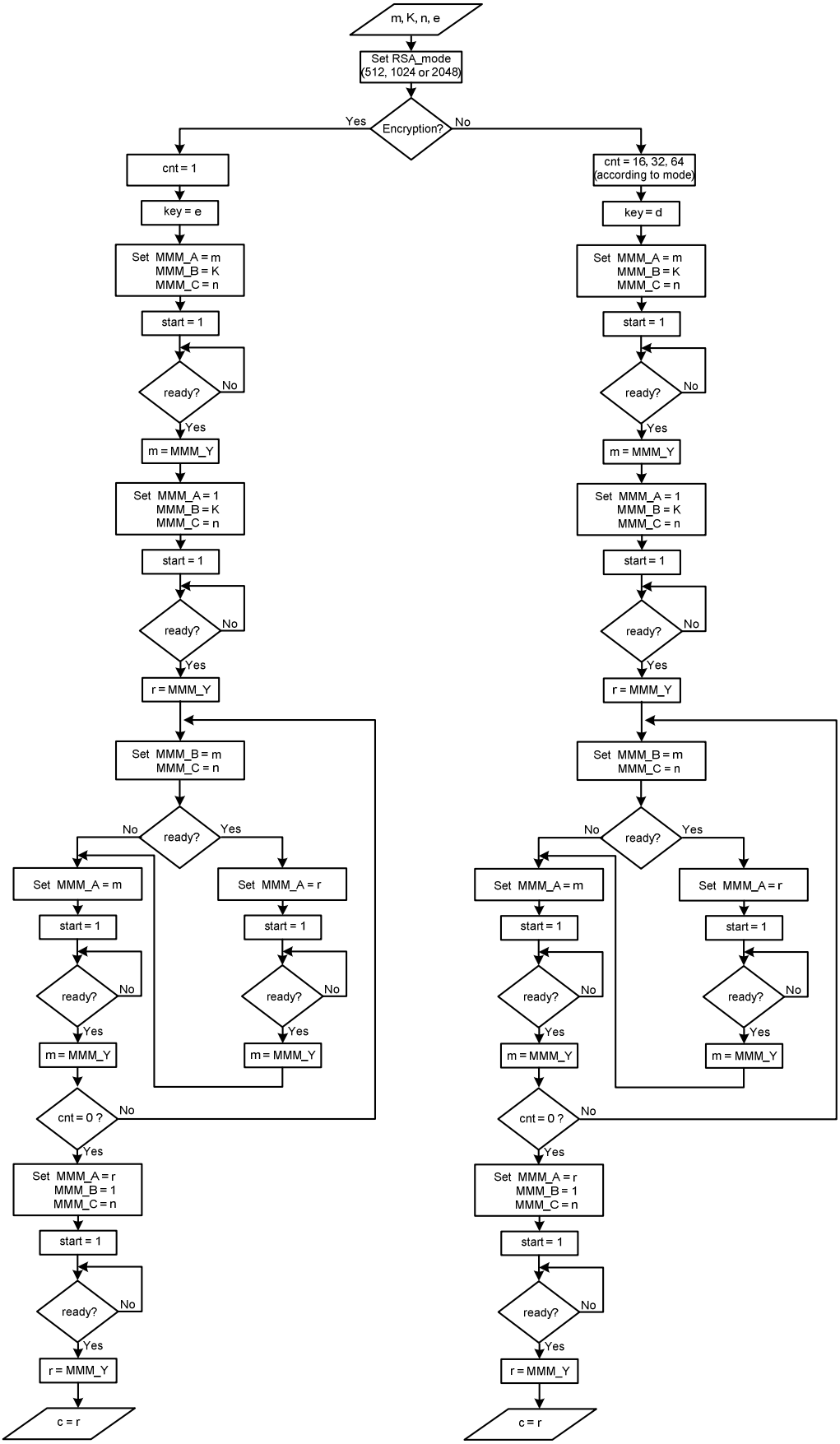

Figure 6.4 Flowchart of RSA.

CHAPTER 7

CONCLUSION

In this thesis, a compact cryptographic processor is implemented. The processor is mainly targeted for IPSec applications, and is composed of a ZPU instruction set compatible microcontroller core, and crytographic coprocessors connected to this core via a simple and generic plug-in interface.

There are three coprocessors capable of implementing the AES encryption and SHA-1 hashing in full compliant with the standards, as well as Montgomery modular multiplication up to 2048-bits. These coprocessors are accessed by the main controller core like regular RAMs, which forms the basis idea for the flexible interface. The interface is generic in the sense that it allows any module to be connected to the main core regardless of the input/output definition or the function of the module with the addition of a simple wrapper around the module.

The cryptographic processor is intended as a proof-of-concept for the flexible interface and a development platform for a commercial IPSec product. It will be possible to evaluate performance of the complete IPSec protocol suite on this processor on either simulation or FPGA development boards.

In order to verify this claim and demonstrate the suitability of the processor design, a set of IPSec components and protocols (RSA, AES-CCM and HMAC-SHA-1) are written in C, compiled to generate ZPU machine code, and run on the processor at simulation level.

This chapter proceeds with a summary of overall results of this research, followed by the suggestions directions for future work.

7.1 Results

To the best of our knowledge, this is the first cryptographic processor that combines AES, SHA-1 and MMM cores in a single design, and provides a flexible interface that allows integration of even more modules of any kind. The closest solution we have come across is the latest Intel

processor family which has custom AES instructions. However, they are not only too expensive for embedded applications, but also lack SHA-1 and fast modular multiplication capabilities.

The resultant processor occupies only 1982 slices and can fit into the smallest Xilinx Virtex-5 device. The maximum clock frequency achievable on this device is 63.41 MHz. The limiting factor for the speed is the AES data path. It is possible to further improve the speed by simply implementing AES substitution boxes with ROM or RAM based lookup tables instead of the finite field arithmetic circuitry in the present design. On the other hand, this type of implementation offers a technology independent design.

The cryptographic processor shows superior throughput performance compared to regular embedded processors (such as MIPS or IA-32 based architectures), thanks to the maximum possible performance achieved by the fully parallel AES and SHA-1 cores. Even the performance of the word-serial modular multiplier core can be considered to be higher than most designs in terms of through per slice.

There is, of course, considerable throughput loss with respect to custom hardware solutions. The average AES performance of the processor is almost 20 percent of that of the AES coprocessor when run in standalone mode. This is a direct result of software overhead, which adds several extra cycles during memory transfers. However it is an acceptable cost considering the flexibility and reconfigurability of the system.

7.2 Directions for Future Work

Two separate directions are envisioned for the future of the present design. The first is to improve the microcontroller core, which can be achieved simply by implementing more instructions in hardware instead of emulation. Such an effort will considerably decrease the program memory size thereby also decreasing the total system area, and increase the throughput by minimizing time in the execution of complex instruction. The effect on the area and overall speed of the system will be marginal since the main bottlenecks for both the area and speed performance come from the coprocessors. Processor performance can be further improved with the introduction of a pipeline into the system, which can drop the cycle count per instruction significantly. However it will result in additional complexity, such as use of dual-port memories, additional registers for temporary storage, and instruction pre-fetch logic. Such a solution may not actually be worth the effort.

The next improvement will be in the speed-up of memory transfers. This can be achieved via a simple direct memory access engine embedded into the processor as an additional accelerator. With this approach, instead of transferring data word-by-word within a for loop, it will be

possible to transfer blocks of data with a single instruction. For example, in the current design, the worst case AES encryption (AES-256) takes 15 cycles to complete. However, writing the 4 words of data to the AES input registers and reading the result back from the takes up to 48 cycles. This is more than 3 times the actual processing time.

In a direct memory access engine solution, the first alternative would be to build such engine into each coprocessor separately. It will then be sufficient to provide the start pointers for the input data and the result to this engine together with a start signal (write into the virtual CSR as before). The engine will read the 4 words of data from memory in 4 cycles, in parallel write them to AES input registers, and increment the start pointer by 1 at every read, start the AES engine, upon completion of encryption read the results from the AES output registers and transfer them to the memory again in 4 cycles. The total number of cycles can be as low as 12, which is only 25 per cent of the current number.

It can be further improved by implementing a nested DMA engine, which writes the outputs of the previous run to and reads the inputs of the next run from the memory while the current AES run is still in progress. This will require double buffering, but effectively eliminate the memory transfer overhead, at least for the coprocessor runs.

In a second approach, there can be a single DMA engine that serves all coprocessors. This solution will not be as effective as individual DMA engines. However, it will be much more compact and configurable.

The best approach is to determine the system requirements with respect to the master application, and implement the solutions that will address these requirements. The first step for this approach is to tabulate the improvement and cost of each solution, which in fact is the main future work to be done.

As a further step, the design and implementation of coprocessors for elliptic curve operations and other widely used crypto algorithms (such as Camellia [68], TDES [49,69] and the upcoming new hash standard [70]) can be considered. They can be plug into the present architecture to come up with a universal cryptographic processor.

REFERENCES

[1] Liddell and Scott's Greek-English Lexicon, Oxford University Press, 1984

[2] *Cryptography*, http://en.wikipedia.org/wiki/Cryptography

[3] *Crypto History*,

http://www.cryptool.de/index.php/en/crypto-history-documentationmenu-54.html

[4] Reinke, E. C., *Classical Cryptography*, The Classical Journal 58 (3), December 1992.

[5] Kahn, D. , *The Codebreakers: The Story of Secret Writing*, Macmillan New York, 1967.

[6] Meiser, G., *Efficient Implementation of Stream Ciphers on Embedded Processors*, M.Sc.Thesis, Ruhr-University Bochum, Germany, 2007

[7] Aydos, M., Sunar, B., Koc, C. K., *An Elliptic Curve Cryptography based Authentication and Key Agreement Protocol for Wireless Communication*, 2nd International Workshop on Discrete Algorithms and Methods for Mobile Computing and Communications, Texas, 1998.

[8] Crowe, F., Daly A., Kerins T., Marnane, W., *Single-Chip FPGA Implementation of a Cryptographic Co-Processor*, Proceedings of International IEEE Conference on Field-Programmable Technology, 2004.

[9] Mangard, S., Aigner, M., Dominikus, S., *A Highly Regular and Scalable AES Hardware Architecture*, IEEE Transactions on Computers, April 2003.

[10] Kang, Y., Kim, D.W., Kwon, T.W. , Choi, J.R., *An efficient implementation of hash function processor for IPSEC*, Proceedings of Third IEEE Asia-Pacific Conference on ASIC, 2002.

[11] Smith, M. J. S., *Application-Specific Integrated Circuits*, Addison-Wesley VLSI Systems Series, 1997.

[12] Wolf, W., *FPGA-Based System Design*, Prentice Hall, 2004.

[13] Paar, C., *A New Architecture for A Parallel Finite Field Multiplier with Low Complexity Based On Composite Fields*, IEEE Transactions on Computers, 1996.

[14] Carlson, D., Brasili, D. Hughes, A., Jain, A., Kiszely, T., Kodandapani, P., Vardharajan, A., Xanthopoulos, T., Yalala, V., *A High Performance SSL IPSEC Protocol Aware Security Processor*, IEEE Solid-State Circuits Conference, Digestt of Technical Papers, 2003.

[15] Su, C., Wang C., Cheng, K., Huang, C., Wu, C., *Design and test of a scalable security processor*, Prooceedings of the IEEE Design Automation Conference, 2005.

[16] *Intel Advanced Encryption Standard (AES) Instructions Set*, Intel Mobility Group, Israel Development Center, Israel, 2010.

[17] *ZPU*, http://repo.or.cz/w/zpu.git?a=blob_plain;f=zpu/docs/zpu_arch.html

[18] *Instruction Set*, http://en.wikipedia.org/wiki/Instruction_set

[19] PKCS #1 v2.1: *RSA Cryptography Standard*.

[20] *Advanced Encryption Standard*, FIPS PUB 197, 2001.

[21] *Secure Hash Standard*, FIPS 180-2, 2002.

[22] Sklavos, N., *On the Hardware Implementation Cost of Crypto-Processors Architectures*, Information Security Journal: A Global Perspective, 19:53–60, 2010

[23] Hodjat, A., Schaumont, P., Verbauwhede, I., *Architectural Design Features of a Programmable High Throughput AES Coprocessor*, Prooceedings of the International Conference on Information Technology: Coding and Computing, 2004.

[24] Somani, A., Faisal, T., Mohammad I. K., *High Performance Elliptic Curve GF (2m) Crypto-Processor*, Information Technology Journal, vol.5, pp.742-748, 2006.

[25] Kim, H.W., Lee, S., *Design and implementation of a private and public key crypto processor and its application to a security system*, IEEE Transactions on Consumer Electronics, Vol.50, pp.214-224, 2004.

[26] Koschuch et al., *Hardware/Software Co-Design of Elliptic Curve Cryptography on an 8051 Microcontroller*, Proceedings of CHES 2006, LNCS 4249, pp. 430–444, 2006.

[27] Buchty, R., Heintze, N., Oliva, D., *Cryptonite – A Programmable Crypto Processor Architecture for High-Bandwidth Applications*, Lecture Notes in Computer Science, Vol.2981, pp.184-198, 2004.

[28] *Complex Instruction Set Computer*,

http://en.wikipedia.org/wiki/Complex_instruction_set_computer

[29] *VLSI IP Cores*, http://www.unistring.com/faqsfive.htm

[30] *Reduced Instruction Set Computer*,

http://en.wikipedia.org/wiki/Reduced_instruction_set_computer

[31] Null, L., Lobur, J., *The Essentials of Computer Organization and Architecture*, Jones and Bartlett Publishers, 2003.

[32] *GCC, the GNU Compiler Collection*, http://gcc.gnu.org/

[33] Miyamoto, A., Homma, N., Aoki, T., Satoh, A., *Systematic design of high-radix Montgomery multipliers for RSA processors*, IEEE International Conference on Computer Design, 2008.

[34] Hong, J. H., Li, W. J., *A Novel and Scalable RSA Cryptosystem Based on 32-Bit Modular Multiplier*, IEEE Computer Society Annual Symposium on VLSI, 2008.

[35] Amanor, D. N., *Efficient Hardware Architectures for Modular Multiplication*, M.Sc. Thesis, 2005.

[36] Hsieh, Y. H., *Design and Implementation of an RSA Encryption/Decryption Processor on IC Smart Card*, M.Sc. Thesis, National Taiwan University, Taiwan, 1999.

[37] *IPSec*, http://en.wikipedia.org/wiki/IPsec

[38] *The Keyed-Hash Message Authentication Code*, FIPS PUB 198, 2002.

[39] *Counter with CBC-MAC (CCM)*, RFC 3610, 2003.

[40] *Transmission Control Protocol*,
http://en.wikipedia.org/wiki/Transmission_Control_Protocol

[41] *IPv4*, http://en.wikipedia.org/wiki/IPv4

[42] *IPv6*, http://en.wikipedia.org/wiki/IPv6

[43] Kozierok, C. M., *The TCP/IP Guide*, 2005.

[44] *RFC Index*, http://tools.ietf.org/rfc/index

[45] *MIPS32 Architecture*, MIPS Technologies.

[46] *Side Channel Attack*, http://en.wikipedia.org/wiki/Side_channel_attack

[47] *The Side Channel Cryptanalysis Lounge*,

http://www.crypto.ruhr-uni-bochum.de/en_sclounge.html

[48] Schwartz, J., *U.S. Selects a New Encryption Technique*, New York Times, October 3, 2000.

[49] *Data Encryption Standard*, FIPS 46-3, 1999.

[50] *Substitution-permutation Network*, http://en.wikipedia.org/wiki/Substitution-permutation_network

[51] *Feistel Cipher*, http://en.wikipedia.org/wiki/Feistel_cipher

[52] Paar, C., *Efficient VLSI Architectures for Bit-Parallel Computation in Galois Fields*, Ph.D. Thesis, Univ. of Essen, Düsseldorf, 1994.

[53] Dandalis, A., Prasanna, V.K., Rolim, J.D.P., *A Comparative Study of Performance of AES Candidates Using FPGAs*, The Third Advanced Encryption Standard (AES3) Candidate Conference, 13-14 April 2000, New York, USA.

[54] *Schneier on Security: Cryptanalysis of SHA-1*, http://www.schneier.com/blog/archives/2005/02/cryptanalysis_o.html

[55] Grembowski et al., *Comparative Analysis of the Hardware Implementations of Hash Functions SHA-1 and SHA-512*, ISC 2002 Proceedings, LNCS 2433, pp. 75–89, 2002.

[56] *Modular Exponentiation Core Family for Xilinx FPGA*, Full Datasheet, Helion Technology.

[57] McLoone, M., McCanny J., *Single-chip FPGA Implementation of the Advanced Encryption Standard Algorithm*, Field-Programmable Logic and Applications, Springer, 2001.

[58] Leong, P. H. W., Leung, I. K. H., *A Microcoded Elliptic Curve Processor Using FPGA Technology*, IEEE Trans. Very Large Scale Integr. (VLSI) Syst., vol. 10, no. 5, pp. 550-559, Oct. 2002.

[59] Thull, D., Sannino, R., *Performance considerations for an embedded implementation of OMA DRM 2*, Design, Automation and Test in Europe, 2005.

[60] Gupta, V., Gupta, S., Chang, S., *Performance Analysis of Elliptic Curve Cryptography for SSL*, Proc. of the ACM Workshop on Wireless Security, ACM Press, Atlanta (GA), USA, 2002.

[61] Osvik, D. A., Bos, J. W., Stefan, D., Canright, D., *Fast software AES encryption*, In International Workshop on Fast Software Encryption, LNCS Springer, 2010.

[62] *Using Advanced Encryption Standard (AES) Counter Mode*, RFC3686, 2004.

[63] *CBC-MAC*, http://en.wikipedia.org/wiki/CBC-MAC

[64] *Using Advanced Encryption Standard (AES) CCM Mode with IPsec Encapsulating Security Payload (ESP)*, RFC4309, 2005.

[65] American Bankers Association, *Keyed Hash Message Authentication Code*, ANSI X9.71, Washington, D.C., 2000.

[66] Krawczyk, H., Bellare, M., Canetti, R., *HMAC: Keyed-Hashing for Message Authentication*, Internet Engineering Task Force, Request for Comments (RFC) 2104, February 1997.

[67] *The Use of HMAC-SHA-1-96 within ESP and AH*, RFC2404, 1998.

[68] Aoki, K., Ichikawa, T., Kanda, M., Matsui, M., Moriai, S., Nakajima J., Tokita, T., *Camellia: a 128-bit block cipher suitable for multiple platforms – design and analysis*, Lecture Notes in Computer Science, Vol. 2012, 2001, pp. 39-56.

[69] ANSI X9.52: *Triple Data Encryption Algorithm Modes of Operation*.

[70] *SHA-3*, http://en.wikipedia.org/wiki/SHA-3

APPENDIX A

VERILOG CODES OF PROCESSOR AND COPROCESSORS

```
`timescale  1 ns / 1 ns
////////////////////////////////////////////////
//
// ZPU Processor Top Block
//
module  zpu_top
(ck, rn , ext_sel , ext_adr , ext_wen , ext_ren , ext_inp , ext_rdy
,ext_out) ;


input            ck         ; // Rising edge clock
input            rn         ; // Active low reset
input            ext_sel    ;
input   [15:0]   ext_adr    ;
input            ext_wen    ;
input            ext_ren    ;
input   [31:0]   ext_inp    ; // Data input from CPU

output           ext_rdy    ; // Ready output to CPU
output  [31:0]   ext_out    ; // Data output to CPU

wire    [31:0]   out        ; // RAM output
wire             busy       ;
wire    [15:0]   adr        ;
wire             wen        ;
wire             ren        ;
wire    [31:0]   inp        ;
wire    [2:0]    alu_op_sel ;
wire    [7:0]    inst       ;
wire             im_flag    ;
wire             emu_flag   ;
wire             ssp_flag   ;
wire             lsp_flag   ;
wire             asp_flag   ;
wire    [6:0]    im_data    ;
wire    [4:0]    emu_data   ;
wire    [4:0]    ssp_data   ;
wire    [4:0]    lsp_data   ;
wire    [3:0]    asp_data   ;
wire             oth_flag   ;
wire             psh_flag   ;
wire             ppc_flag   ;
wire             add_flag   ;
wire             and_flag   ;
wire             or_flag    ;
wire             ld_flag    ;
wire             not_flag   ;
wire             flp_flag   ;
wire             nop_flag   ;
```

```
wire             str_flag   ;
wire             pop_flag   ;
wire             pmem_re    ;
wire             pc_load    ;
wire             pc_inc     ;
wire             sp_load    ;
wire             sp_inc     ;
wire             sp_dec     ;
wire             tmp_ld_sp  ;
wire             tmp_ld_rd  ;
wire             idim_upd   ;
wire    [2:0]    cyc_in     ;
wire             act_in     ;
wire    [15:0]   sp_in      ;
wire    [15:0]   pc_in      ;
wire    [31:0]   tmp_in     ;
wire             idim_in    ;
wire             rdy_in     ;
wire             busy_ext  =  busy  ||  ext_sel ;

reg     [31:0]   alu_out    ;
reg     [2:0]    cyc        ;
reg              act        ;
reg     [15:0]   sp         ;
reg     [15:0]   pc         ;
reg     [31:0]   tmp        ;
reg              idim       ;
reg              rdy        ;

zpu_pmem u_zpu_pmem (
   .pmem_re ( pmem_re ) ,
   .pc      ( pc      ) ,
   .inst    ( inst    ) ,
   .ck      ( ck      )
) ;

assign im_flag = inst[7]   ; // immediate instruction flag
assign im_data = inst[6:0] ; // immediate data
assign emu_flag = (inst[7:5]==3'b001 ) ; // emulate instruction flag
assign ssp_flag = (inst[7:5]==3'b010 ) ; // storesp instruction flag
assign lsp_flag = (inst[7:5]==3'b011 ) ; // loadsp instruction flag
assign asp_flag = (inst[7:4]==4'b0001) ; // addsp instruction flag
assign emu_data = inst[4:0] ; // emulate data
assign ssp_data = inst[4:0] ; // storesp data
assign lsp_data = inst[4:0] ; // loadsp data
assign asp_data = inst[3:0] ; // addsp data
assign oth_flag = (inst[7:4]==4'b0000) ; // other instructions flag
assign psh_flag = (oth_flag&&(inst[3:0]==4'b0010)) ; // pushsp flag
assign ppc_flag = (oth_flag&&(inst[3:0]==4'b0100)) ; // poppc flag
assign add_flag = (oth_flag&&(inst[3:0]==4'b0101)) ; // add flag
assign and_flag = (oth_flag&&(inst[3:0]==4'b0110)) ; // and flag
assign or_flag  = (oth_flag&&(inst[3:0]==4'b0111)) ; // or flag
assign ld_flag  = (oth_flag&&(inst[3:0]==4'b1000)) ; // load flag
assign not_flag = (oth_flag&&(inst[3:0]==4'b1001)) ; // not flag
assign flp_flag = (oth_flag&&(inst[3:0]==4'b1010)) ; // flip flag
assign nop_flag = (oth_flag&&(inst[3:0]==4'b1011)) ; // nop flag
assign str_flag = (oth_flag&&(inst[3:0]==4'b1100)) ; // store flag
assign pop_flag = (oth_flag&&(inst[3:0]==4'b1101)) ; // popsp flag

assign act_in = (cyc == 3'b000) ? 1'b1 : act ;
```

```
always  @ ( posedge ck  or  negedge rn )
  if   ( !rn ) act <= #1 0        ;
  else         act <= #1 act_in   ;

assign cyc_in = (cyc == 3'b000)    ? 3'b001            : (
                (act && !busy_ext) ? {cyc[1:0],cyc[2]} : cyc) ;

always  @ ( posedge ck  or  negedge rn )
  if   ( !rn ) cyc <= #1 0        ;
  else         cyc <= #1 cyc_in   ;

assign pmem_re = (cyc==3'b000) || ( act && (~busy_ext) && cyc[2] ) ;

assign pc_inc  = cyc[0] ;

assign pc_load = cyc[1] && (ppc_flag || emu_flag) ;

assign pc_in = (act && !busy_ext) ? (pc_load ? out:( pc_inc ?
pc+1:pc )) : pc ;

always  @ ( posedge ck  or  negedge rn )
  if   ( !rn ) pc <= #1 0         ;
  else         pc <= #1 pc_in     ;

assign sp_inc=cyc[0] ?
(ppc_flag||add_flag||and_flag||or_flag||str_flag):(
             cyc[1] ? (ssp_flag||str_flag)
:1'b0);

assign sp_dec=cyc[0] ?
(lsp_flag||psh_flag||emu_flag||(im_flag&&(idim==0))) :
                       1'b0 ;

assign sp_load = cyc[1] && pop_flag ;

assign sp_in = (act && !busy_ext) ? (
                sp_load           ? out  : (
                sp_inc            ? sp+1 : (
                sp_dec            ? sp-1 : sp) ) ) : sp ;

always  @ ( posedge ck  or  negedge rn )
  if   ( !rn ) sp <= #1 2039      ;
  else         sp <= #1 sp_in     ;

assign tmp_ld_sp = cyc[0] && psh_flag ;

assign tmp_ld_rd = cyc[1] &&
(asp_flag||add_flag||and_flag||or_flag||str_flag);

assign tmp_in=(act && !busy_ext) ? (tmp_ld_rd ? out:(tmp_ld_sp ?
sp:tmp)):tmp ;

always  @ ( posedge ck  or  negedge rn )
  if   ( !rn ) tmp <= #1 0        ;
  else         tmp <= #1 tmp_in   ;

assign idim_upd = ( im_flag && cyc[1] ) ;

assign idim_in=(act && !busy_ext) ? (idim_upd ? 1:(cyc[1] ?
0:idim)):idim ;
```

```
always  @ ( posedge ck  or  negedge rn )
  if   ( !rn ) idim <= #1 0       ;
  else         idim <= #1 idim_in ;

zpu_ram_top  u_zpu_ram_top (
   .ck      ( ck       ) ,
   .rn      ( rn       ) ,
   .wen     ( wen      ) ,
   .ren     ( ren      ) ,
   .adr     ( adr      ) ,
   .inp     ( inp      ) ,
   .ext_sel ( ext_sel  ) ,
   .ext_wen ( ext_wen  ) ,
   .ext_ren ( ext_ren  ) ,
   .ext_adr ( ext_adr  ) ,
   .ext_inp ( ext_inp  ) ,
   .busy    ( busy     ) ,
   .out     ( out      ) ,
   .ext_out ( ext_out  )
 ) ;

assign wen=cyc[1] ?

(emu_flag||not_flag||flp_flag||psh_flag||ssp_flag||lsp_flag||im_flag
):(
           cyc[2] ?
        (asp_flag||add_flag||and_flag||or_flag||ld_flag||str_flag) :
         1'b0) ;
assign ren=cyc[0] ?
(emu_flag||str_flag||pop_flag||not_flag||flp_flag||add_flag||
and_flag||or_flag||ld_flag||ppc_flag||asp_flag||ssp_flag||lsp_flag||
(im_flag&&(idim==1))) : (
           cyc[1] ?
(asp_flag||add_flag||and_flag||or_flag||ld_flag||str_flag):1'b0) ;

wire asp = !( (cyc[2]&&str_flag) || (cyc[1]&&ld_flag) ||
(cyc[1]&&ssp_flag) ||
              (cyc[1]&&asp_flag) || (cyc[0]&&lsp_flag)||
(cyc[0]&&emu_flag) ) ;

assign adr = ( ( cyc[2] && str_flag ) ? tmp[15:0]     : 0 )  |
             ( ( cyc[1] && ld_flag  ) ? out[15:0]     : 0 )  |
             ( ( cyc[1] && ssp_flag ) ? (sp+ssp_data) : 0 )  |
             ( ( cyc[1] && asp_flag ) ? (sp+asp_data) : 0 )  |
             ( ( cyc[0] && lsp_flag ) ? (sp+lsp_data) : 0 )  |
             ( ( cyc[0] && emu_flag ) ? alu_out[15:0] : 0 )  |
             (   asp                  ? sp            : 0 )  ;

assign inp = ( (cyc[2] && (asp_flag || add_flag || and_flag ||
or_flag)) || (cyc[1] && (not_flag || flp_flag ||
((idim==1)&&im_flag))) || (cyc[0] && emu_flag) )
? alu_out : (
             (  cyc[1] && ((idim==1)&&im_flag) )
? im_data : (
             (  cyc[1] && psh_flag )
? tmp     : (
             (  cyc[1] && emu_flag )
? pc      : out ) ) ) ;

assign alu_op_sel = im_flag                ? 3'd5 : (
                    (asp_flag || add_flag) ? 3'd0 : (
```

```
                  and_flag               ? 3'd1 : (
                  or_flag                ? 3'd2 : (
                  not_flag               ? 3'd3 : (
                  flp_flag               ? 3'd4 : (
                  emu_flag               ? 3'd6 : 3'd7 ) ) ) ) ) ) ;

always @ (*)
      case (alu_op_sel)
      0       : alu_out = out + tmp                   ;
      1       : alu_out = out & tmp                   ;
      2       : alu_out = out | tmp                   ;
      3       : alu_out = ~out                        ;
      4       : alu_out = {
out[0],out[1],out[2],out[3],out[4],out[5],out[6],out[7],out[8],out[9
],out[10],out[11],out[12],out[13],out[14],out[15],out[16],out[17],ou
t[18],out[19],out[20],out[21],out[22],out[23],out[24],out[25],out[26
],out[27],out[28],out[29],out[30],out[31] } ;
      5       : alu_out = { out[24:0] , im_data     } ;
      6       : alu_out = { 22'd0 , emu_data , 5'd0 } ;
      default : alu_out = 0                           ;
      endcase

assign rdy_in = ext_sel ? 0 : ((wen && (adr == 16'h0FFF)) ? 1 : rdy)
;

always  @ ( posedge ck  or  negedge rn )
  if  ( !rn )  rdy  <=  0      ;
  else         rdy  <=  rdy_in ;

assign  ext_rdy  =  rdy ;


endmodule
```

```
`timescale  1 ns / 1 ns
////////////////////////////////////////////////
//
// ZPU RAM top block
//
module  zpu_ram_top
       ( ck , rn , adr , wen , ren , inp , ext_sel , ext_adr ,
ext_wen , ext_ren , ext_inp , busy , out , ext_out ) ;


input            ck       ; // Rising edge clock
input            rn       ; // Active low reset
input   [15:0]   adr      ;
input            wen      ;
input            ren      ;
input   [31:0]   inp      ; // Data input
input   [15:0]   ext_adr  ;
input            ext_sel  ;
input            ext_wen  ;
input            ext_ren  ;
input   [31:0]   ext_inp  ; // Data input from CPU

output           busy     ; // Busy signal to ZPU
output  [31:0]   out      ; // Data output
output  [31:0]   ext_out  ; // Data output to CPU
```

```
reg     [3:0]    sel      ;
reg     [31:0]   outi     ;
wire    [31:0]   int0     ;
wire    [31:0]   int1     ;
wire    [31:0]   int2     ;
wire    [31:0]   int3     ;
wire    [8:0]    zadr     ;
wire             zwen     ;
wire             zren     ;
wire    [31:0]   zinp     ;
wire             absy , sbsy , mbsy ;

assign zwen = ext_sel ? ext_wen : wen ;
assign zren = ext_sel ? ext_ren : ren ;
assign zadr = ext_sel ? ext_adr[10:2] : adr[10:2] ;
assign zinp = ext_sel ? ext_inp : inp ;

zpu_ram  u_zpu_ram (
   .ck   ( ck   ) ,
   .wen  ( zwen ) ,
   .ren  ( zren ) ,
   .adr  ( zadr ) ,
   .inp  ( zinp ) ,
   .out  ( int0 )
 ) ;

aes_128  u_aes_ram (
   .ck   ( ck   ) ,
   .rn   ( rn   ) ,
   .wen  ( wen  ) ,
   .ren  ( ren  ) ,
   .adr  ( adr  ) ,
   .inp  ( inp  ) ,
   .out  ( int1 ) ,
   .busy ( absy )
 ) ;

sha1     u_sha_ram (
   .ck   ( ck   ) ,
   .rn   ( rn   ) ,
   .wen  ( wen  ) ,
   .ren  ( ren  ) ,
   .adr  ( adr  ) ,
   .inp  ( inp  ) ,
   .out  ( int2 ) ,
   .busy ( sbsy )
 ) ;

mmm_top  u_mmm_ram (
   .ck   ( ck   ) ,
   .rn   ( rn   ) ,
   .wen  ( wen  ) ,
   .ren  ( ren  ) ,
   .adr  ( adr  ) ,
   .inp  ( inp  ) ,
   .out  ( int3 ) ,
   .busy ( mbsy )
 ) ;

always  @ ( posedge ck  or  negedge rn )
```

```
  if      ( !rn   )  sel  <=  0          ;
  else               sel  <=  adr[15:12] ;

always @ (*)
      case (sel)
      0       : outi = int0 ;
      1       : outi = int1 ;
      2       : outi = int2 ;
      3       : outi = int3 ;
      default : outi = 0    ;
      endcase

assign busy    = absy | sbsy | mbsy ;

assign out     = outi ;

assign ext_out = int0 ;


endmodule
```

```
////////////////////////////////////////////////
// Top level module for AES
//
module  aes_128  ( adr , wen , ren , inp , ck , rn , busy , out ) ;


input   [15:0]   adr  ;
input            wen  ;
input            ren  ;
input   [31:0]   inp  ; // Data input
input            ck   ; // Rising edge clock
input            rn   ; // Active low reset

output           busy ;
output  [31:0]   out  ; // Data output

wire    [2:0]    kwadr  =  adr[2:0]                        ;
wire             kwen   =  wen  &&  ( adr[15:8] == 8'h12 ) ;
wire    [1:0]    dwadr  =  adr[1:0]                        ;
wire             dwen   =  wen  &&  ( adr[15:8] == 8'h10 ) ;
wire    [1:0]    dradr  =  adr[1:0]                        ;
wire             dren   =  ren  &&  ( adr[15:8] == 8'h11 ) ;
wire             mwen   =  wen  &&  ( adr[15:8] == 8'h13 ) ;
wire             cwen   =  wen  &&  ( adr[15:8] == 8'h1F ) ;
wire             ready     ;
wire             startp    ;
wire             round0    ;
wire             round1    ;
wire             roundfn   ;
wire             round147  ;
wire             round369  ;
wire             round_odd ;
wire             rcon_upd  ;
wire             active    ;
wire    [0:127]  keyr      ;
wire    [0:127]  sreg      ;
wire    [0:255]  kreg      ; // Sreg and Kreg outputs
wire    [0:7]    rcon      ; // Rcon register output
```

```
wire    [0:127]  sint      ;
wire    [0:7]    rint      ; // Internal Rcon value
reg              start     ;
reg     [1:0]    mode      ; // Key mode
                             // (mode=(00)-128/mode=(01)-192/
                             //  mode=(10)-256)
reg     [0:127]  outi      ;
reg     [0:31]   pinp      ;
reg     [0:31]   outf      ;

always  @ ( posedge ck  or  negedge rn )
  if      ( !rn  )    mode  <= #1  0        ;
  else if ( mwen )    mode  <= #1  inp[1:0] ;

always  @ ( posedge ck  or  negedge rn )
  if      ( !rn   )    start  <= #1  0 ;
  else if ( ready )    start  <= #1  0 ;
  else if ( cwen  )    start  <= #1  1 ;

pulse_gen  u_start  (
  .pulse_in   ( start  ) ,
  .pulse_out  ( startp ) ,
  .pulse_type ( 1'b0   ) ,
  .ck         ( ck     ) ,
  .rn         ( rn     )
 ) ;
aes_control  u_control  (
  .start     ( startp    ) ,
  .ck        ( ck        ) ,
  .rn        ( rn        ) ,
  .mode      ( mode      ) ,
  .round0    ( round0    ) ,
  .round1    ( round1    ) ,
  .roundfn   ( roundfn   ) ,
  .round147  ( round147  ) ,
  .round369  ( round369  ) ,
  .round_odd ( round_odd ) ,
  .rcon_upd  ( rcon_upd  ) ,
  .active    ( active    ) ,
  .ready     ( ready     )
) ;

aes_sreg  u_sreg  (
  .wadr   ( dwadr  ) ,
  .wen    ( dwen   ) ,
  .active ( active ) ,
  .inp    ( inp    ) ,
  .state  ( sint   ) ,
  .ck     ( ck     ) ,
  .rn     ( rn     ) ,
  .out    ( sreg   )
) ;

key_unit  u_kunit  (
  .wadr      ( kwadr     ) ,
  .wen       ( kwen      ) ,
  .ck        ( ck        ) ,
  .rn        ( rn        ) ,
  .inp       ( inp       ) ,
  .rc        ( rcon      ) ,
  .mode      ( mode      ) ,
```

```
  .start     ( startp    ) ,
  .active    ( active    ) ,
  .round0    ( round0    ) ,
  .round1    ( round1    ) ,
  .round147  ( round147  ) ,
  .round369  ( round369  ) ,
  .round_odd ( round_odd ) ,
  .krout     ( keyr      ) ,
  .out       ( kreg      )
) ;

aes_rcon  u_rcon  (
  .start    ( startp   ) ,
  .rcon_upd ( rcon_upd ) ,
  .rc       ( rint     ) ,
  .ck       ( ck       ) ,
  .rn       ( rn       ) ,
  .out      ( rcon     )
) ;

aes_comb  u_comb  (
  .sreg      ( sreg      ) ,
  .kreg      ( kreg      ) ,
  .rcon      ( rcon      ) ,
  .keyr      ( keyr      ) ,
  .round0    ( round0    ) ,
  .roundfn   ( roundfn   ) ,
  .active    ( active    ) ,
  .state     ( sint      ) ,
  .rc        ( rint      )
) ;

always  @ ( posedge ck  or  negedge rn )
  if      ( !rn   )  outi  <= #1  0    ;
  else               outi  <= #1  sint ;

always @ (*)
      case (dradr[1:0])
      0       : pinp = outi[0  : 31] ;
      1       : pinp = outi[32 : 63] ;
      2       : pinp = outi[64 : 95] ;
      default : pinp = outi[96 :127] ;
      endcase

always  @ ( posedge ck  or  negedge rn )
  if      ( !rn   )  outf  <= #1  0    ;
  else if ( dren  )  outf  <= #1  pinp ;

assign busy = active ;

assign out = outf ;


endmodule
```

```
////////////////////////////////////////////////
// Top level module for SHA-1
//
module  sha1  ( adr , wen , ren , inp , ck , rn , busy , out ) ;
```

```
input   [15:0]   adr   ;
input            wen   ;
input            ren   ;
input   [31:0]   inp   ; // Data input
input            ck    ; // Rising edge clock
input            rn    ; // Active low reset

output           busy  ;
output  [31:0]   out   ;   // Hash output

wire    [3:0]    dwadr  =  adr[3:0]                          ;
wire             dwen   =  wen  &&  ( adr[15:8] == 8'h20 ) ;
wire    [3:0]    dradr  =  adr[3:0]                          ;
wire             dren   =  ren  &&  ( adr[15:8] == 8'h21 ) ;
wire             mwen   =  wen  &&  ( adr[15:8] == 8'h22 ) ;
wire             cwen   =  wen  &&  ( adr[15:8] == 8'h2F ) ;
wire             clearp, startp ;
wire             cnt79, active, ready ;
wire    [0:1]    sel    ; // Selector
wire    [0:511]  wout   ; // Wreg output
wire    [0:511]  wnxt   ; // Wreg next value
wire    [0:159]  aout   ; // Areg output
wire    [0:159]  anxt   ; // Areg next value
wire    [0:159]  hout   ; // Hreg output
wire    [0:159]  hnxt   ; // Hreg next value
wire    [0:31]   outf   ;
reg              start, clear ;

always  @ ( posedge ck  or  negedge rn )
  if  ( !rn )          clear  <=  0 ;
  else if ( ready )    clear  <=  0 ;
  else if ( mwen  )    clear  <=  1 ;

always  @ ( posedge ck  or  negedge rn )
  if  ( !rn )          start  <=  0 ;
  else if ( ready )    start  <=  0 ;
  else if ( cwen  )    start  <=  1 ;

pulse_gen  u_start  (
  .pulse_in   ( start  ) ,
  .pulse_out  ( startp ) ,
  .pulse_type ( 1'b0   ) ,
  .ck         ( ck     ) ,
  .rn         ( rn     )
 ) ;

pulse_gen  u_clear  (
  .pulse_in   ( clear  ) ,
  .pulse_out  ( clearp ) ,
  .pulse_type ( 1'b0   ) ,
  .ck         ( ck     ) ,
  .rn         ( rn     )
 ) ;

sha1_control  u_control  (
  .start   ( startp ) ,
  .ck      ( ck     ) ,
  .rn      ( rn     ) ,
  .cnt79   ( cnt79  ) ,
```

```
  .sel     ( sel    ) ,
  .active  ( active ) ,
  .ready   ( ready  )
) ;

sha1_wreg  u_wreg  (
  .wadr   ( dwadr  ) ,
  .wen    ( dwen   ) ,
  .active ( active ) ,
  .inp    ( inp    ) ,
  .nxt    ( wnxt   ) ,
  .ck     ( ck     ) ,
  .rn     ( rn     ) ,
  .out    ( wout   )
) ;

sha1_areg  u_areg  (
  .start  ( startp ) ,
  .active ( active ) ,
  .inp    ( hout   ) ,
  .nxt    ( anxt   ) ,
  .ck     ( ck     ) ,
  .rn     ( rn     ) ,
  .out    ( aout   )
) ;

sha1_hreg  u_hreg  (
  .clear ( clearp ) ,
  .load  ( cnt79  ) ,
  .radr  ( dradr  ) ,
  .ren   ( dren   ) ,
  .inp   ( 160'h67452301efcdab8998badcfe10325476c3d2e1f0 ) ,
  .nxt   ( hnxt   ) ,
  .ck    ( ck     ) ,
  .rn    ( rn     ) ,
  .out   ( hout   ) ,
  .outf  ( outf   )
) ;

sha1_comb  u_comb  (
  .ap ( aout[0:31]    ) ,
  .bp ( aout[32:63]   ) ,
  .cp ( aout[64:95]   ) ,
  .dp ( aout[96:127]  ) ,
  .ep ( aout[128:159] ) ,
  .wp ( wout          ) ,
  .hp ( hout          ) ,
  .sel( sel           ) ,
  .an ( anxt[0:31]    ) ,
  .bn ( anxt[32:63]   ) ,
  .cn ( anxt[64:95]   ) ,
  .dn ( anxt[96:127]  ) ,
  .en ( anxt[128:159] ) ,
  .wn ( wnxt          ) ,
  .hn ( hnxt          )
) ;
assign busy = active ;
assign out  =  outf  ;


endmodule
```

```
`timescale  1 ns / 1 ns
////////////////////////////////////////////
//
// Top level module for MMM
//
module  mmm_top ( adr , wen , ren , inp , ck , rn , busy , out ) ;

// Inputs
//
input   [15:0]   adr   ;
input            wen   ;
input            ren   ;
input   [31:0]   inp   ; // Data input
input            ck    ; // Rising edge clock
input            rn    ; // Active low reset

// Outputs
//
output           busy  ;
output  [31:0]   out   ;

// Internal wires and registers
//
wire    [5:0]    a_wadr  =  adr[5:0]                          ;
wire             a_wenb  =  wen   &&  ( adr[15:8] == 8'h30 ) ;
wire    [5:0]    b_wadr  =  adr[5:0]                          ;
wire             b_wenb  =  wen   &&  ( adr[15:8] == 8'h31 ) ;
wire    [5:0]    n_wadr  =  adr[5:0]                          ;
wire             n_wenb  =  wen   &&  ( adr[15:8] == 8'h32 ) ;
wire    [5:0]    dradr   =  adr[5:0]                          ;
wire             drenb   =  ren   &&  ( adr[15:8] == 8'h33 ) ;
wire             mwenb   =  wen   &&  ( adr[15:8] == 8'h34 ) ;
                            //RSA-512(00)/1024(01)/2048(10) enable
wire             cwenb   =  wen   &&  ( adr[15:8] == 8'h3F ) ;
wire                        ready     ;
wire    [5:0]               cycle     ;
wire    [10:0]              loop      ;
wire    [1:0]               phase     ;
wire                        a_renb    ;
wire                        b_renb    ;
wire                        s_renb    ;
wire                        n_renb    ;
wire                        s_wenb    ;
wire                        sbit      ;
wire    [5:0]               a_radr    ;
wire    [5:0]               b_radr    ;
wire    [5:0]               s_radr    ;
wire    [5:0]               n_radr    ;
wire    [5:0]               s_wadr    ;
wire    [31:0]              s_inp     ;
wire                        plc00x    ;
wire                        plc1x     ;
wire                        plc0xx    ;
wire                        plc0x3    ;
wire                        plc13     ;
wire                        plc003    ;
wire                        plc0m0    ;
wire                        plc0x0    ;
wire                        plc23     ;
wire                        plc20     ;
wire                        plc10     ;
```

```
wire                      plc2x     ;
wire                      plc000    ;
wire                      plc1xd    ;
wire                      plc1xdd   ;
wire                      plc0xxd   ;
wire                      plc0xxdd  ;
wire                      plc0x3d   ;
wire                      plc0x3dd  ;
wire                      plc13d    ;
wire                      plc2xd    ;
wire    [34:0]            t         ;
wire                      q         ;
wire                      ao        ;
wire    [1:0]             s_ext     ;
wire    [1:0]             cinp      ;
wire                      arsel     ;
wire    [31:0]            arin      ;
wire    [31:0]            aofull    ;
wire                      qc        ;
wire                      qrin      ;
wire                      crsel     ;
wire    [1:0]             crin      ;
wire    [1:0]             srin      ;
wire                      t0sel     ;
wire                      t0rin     ;
wire                      t0_out    ;
wire                      tmpsel    ;
wire    [30:0]            tmprin    ;
wire    [30:0]            tmp       ;
wire    [31:0]            a_out     ;
wire    [31:0]            b_out     ;
wire    [31:0]            n_out     ;
wire    [31:0]            s_out     ;
wire    [5:0]             sasel     ;
wire                      sesel     ;
wire                      s         ;
wire                      active    ;
reg     [1:0]             rsa_mode  ;
reg                       start     ;
reg     [31:0]            arout     ;
reg                       qrout     ;
reg     [1:0]             crout     ;
reg     [1:0]             srout     ;
reg                       t0rout    ;
reg     [30:0]            tmprout   ;

always  @ ( posedge ck  or  negedge rn )
  if  ( !rn )          rsa_mode  <=  0        ;
  else if ( mwenb  )   rsa_mode  <=  inp[1:0] ;

always  @ ( posedge ck  or  negedge rn )
  if  ( !rn )          start  <=  0 ;
  else if ( ready )    start  <=  0 ;
  else if ( cwenb )    start  <=  1 ;

// Calculations
//
wire [31:0] t0 = ( plc00x  ? 0                                  :
s_out                                ) ;
wire [31:0] t1 = ( plc0xx  ? (q      ? n_out : 0          )  :
(plc1x    ? (~n_out)      : n_out)) ;
```

```
wire [33:0] t2 = ( plc0xx  ? (ao     ? b_out : 0           )  :
(plc13    ? {2'b11,32'd0} : 0    )) ;
wire [33:0] t3 = ( plc0x3  ? (plc003 ? 0     : {s_ext,32'd0}) :
(plc13    ? {s_ext,32'd0} : 0    )) ;
wire [1:0]  t4 = cinp ;
assign  t  =  t0  +  t1  +  t2  +  t3  +  t4 ;

//
assign arsel = plc0m0 ;

assign arin = arsel ? a_out : (plc0x3 ? (arout >> 1) : arout) ;

always  @ ( posedge ck  or  negedge rn )
  if  ( !rn )     arout  <= #1 0       ;
  else            arout  <= #1 arin    ;

assign aofull = arsel ? a_out : arout ;

assign ao = aofull[0] ;

//
assign qc = (plc000 ? 0 : s_out[0]) ^ (ao & b_out[0]) ;

assign qrin = plc0x0 ? qc : qrout ;

always  @ ( posedge ck  or  negedge rn )
  if  ( !rn )     qrout  <= #1 0       ;
  else            qrout  <= #1 qrin    ;

assign q = plc0x0 ? qc : qrout ;

//
assign crsel = !(plc0x3 || plc13 || plc23) ;

assign crin = crsel ? t[33:32] : crout ;

always  @ ( posedge ck  or  negedge rn )
  if  ( !rn )     crout  <= #1 0       ;
  else            crout  <= #1 crin    ;

assign cinp = plc000 ? 2'd0 : (plc10 ? 2'b01 : ((plc0x0 || plc20) ?
0 : crout)) ;

//
assign srin = plc0x3 ? t[34:33] : srout ;

always  @ ( posedge ck  or  negedge rn )
  if  ( !rn )     srout  <= #1 0       ;
  else            srout  <= #1 srin    ;

assign s_ext = srout ;

//
assign sbit = t[33] ;

//
assign t0sel = plc0x3 || plc1x || plc2x ;

assign t0rin = t0sel ? (plc0xx ? t[32] : t[0]) : t0rout ;

always  @ ( posedge ck  or  negedge rn )
```

```
  if  ( !rn )     t0rout  <= #1 0        ;
  else            t0rout  <= #1 t0rin    ;

assign t0_out = (plc0x3d || plc1xd || plc2xd) ? t0rout : t[0] ;

//
assign tmpsel = plc0xx || plc1x || plc2x ;

assign tmprin = tmpsel ? t[31:1] : tmprout ;

always  @ ( posedge ck  or  negedge rn )
  if  ( !rn )     tmprout  <= #1 0         ;
  else            tmprout  <= #1 tmprin    ;

assign tmp = tmprout ;

//
assign s_inp= plc0xxd ? {t0_out,tmp} : {tmp,t0_out} ;

// Modules
//
mmm_control u_control (
   .ck       ( ck       ) ,
   .rn       ( rn       ) ,
   .rsa_mode ( rsa_mode ) ,
   .start    ( start    ) ,
   .sbit     ( sbit     ) ,
   .a_radr   ( a_radr   ) ,
   .b_radr   ( b_radr   ) ,
   .s_radr   ( s_radr   ) ,
   .n_radr   ( n_radr   ) ,
   .s_wadr   ( s_wadr   ) ,
   .a_renb   ( a_renb   ) ,
   .b_renb   ( b_renb   ) ,
   .s_renb   ( s_renb   ) ,
   .n_renb   ( n_renb   ) ,
   .s_wenb   ( s_wenb   ) ,
   .ready    ( ready    ) ,
   .plc1xd   ( plc1xd   ) ,
   .plc1xdd  ( plc1xdd  ) ,
   .plc0xxd  ( plc0xxd  ) ,
   .plc0xxdd ( plc0xxdd ) ,
   .plc0x3d  ( plc0x3d  ) ,
   .plc0x3dd ( plc0x3dd ) ,
   .plc13d   ( plc13d   ) ,
   .plc2xd   ( plc2xd   ) ,
   .plc000   ( plc000   ) ,
   .plc00x   ( plc00x   ) ,
   .plc1x    ( plc1x    ) ,
   .plc0xx   ( plc0xx   ) ,
   .plc0x3   ( plc0x3   ) ,
   .plc13    ( plc13    ) ,
   .plc003   ( plc003   ) ,
   .plc0m0   ( plc0m0   ) ,
   .plc0x0   ( plc0x0   ) ,
   .plc23    ( plc23    ) ,
   .plc20    ( plc20    ) ,
   .plc10    ( plc10    ) ,
   .plc2x    ( plc2x    ) ,
   .active   ( active   )
) ;
```

```
a_tpram u_aram (
   .ck   ( ck     ) ,
   .wen  ( a_wenb ) ,
   .wadr ( a_wadr ) ,
   .wdat ( a_inp  ) ,
   .ren  ( a_renb ) ,
   .radr ( a_radr ) ,
   .rdat ( a_out  )
) ;

b_tpram u_bram (
   .ck   ( ck     ) ,
   .wen  ( b_wenb ) ,
   .wadr ( b_wadr ) ,
   .wdat ( b_inp  ) ,
   .ren  ( b_renb ) ,
   .radr ( b_radr ) ,
   .rdat ( b_out  )
) ;

assign s = active ? 0 : 1 ;

assign sasel = s ? dradr : s_radr ;
assign sesel = s ? drenb : s_renb ;

s_tpram u_sram (
   .ck   ( ck     ) ,
   .wen  ( s_wenb ) ,
   .wadr ( s_wadr ) ,
   .wdat ( s_inp  ) ,
   .ren  ( sesel  ) ,
   .radr ( sasel  ) ,
   .rdat ( s_out  )
) ;

n_tpram u_nram (
   .ck   ( ck     ) ,
   .wen  ( n_wenb ) ,
   .wadr ( n_wadr ) ,
   .wdat ( n_inp  ) ,
   .ren  ( n_renb ) ,
   .radr ( n_radr ) ,
   .rdat ( n_out  )
 ) ;

assign busy = active ;

assign out = s_out ;


endmodule
```

APPENDIX B

C CODES OF APPLICATIONS

```
/*
 *
 * ipsec.h
 * Header file for addresses of IPSec applications
 *
 */

volatile int *ZPU_rsv   ;  ZPU_rsv   = (volatile int*)0x0000 ;
/* Reserved places for ZPU */
volatile int *ZPU_tmp   ;  ZPU_tmp   = (volatile int*)0x0100 ;
/* Temporary registers for processing */
volatile int *CCM_M     ;  CCM_M     = (volatile int*)0x0200 ;
/* CCM ICV length */
volatile int *CCM_lm    ;  CCM_lm    = (volatile int*)0x0210 ;
/* CCM message length */
volatile int *CCM_la    ;  CCM_la    = (volatile int*)0x0220 ;
/* CCM AAD length */
volatile int *CCM_NNCs  ;  CCM_NNCs  = (volatile int*)0x0230 ;
/* salt   ---   CCM nonce */
volatile int *CCM_NNCiv ;  CCM_NNCiv = (volatile int*)0x0240 ;
/* IV     ---   CCM nonce */
volatile int *CCM_AAD   ;  CCM_AAD   = (volatile int*)0x0250 ;
/* CCM AAD */
volatile int *CCM_Kd    ;  CCM_Kd    = (volatile int*)0x0260 ;
/* CCM key input */
volatile int *CCM_conf  ;  CCM_conf  = (volatile int*)0x0270 ;
/* CCM configuration register - Key mode (mode=(00)-128/mode=(01)-
192/mode=(10)-256) */
volatile int *HMAC_lm   ;  HMAC_lm   = (volatile int*)0x0300 ;
/* HMAC message length */
volatile int *HMAC_Kd   ;  HMAC_Kd   = (volatile int*)0x0310 ;
/* HMAC key input */
volatile int *RSA_e     ;  RSA_e     = (volatile int*)0x0400 ;
/* RSA e public key */
volatile int *RSA_d     ;  RSA_d     = (volatile int*)0x0410 ;
/* RSA d private key */
volatile int *RSA_N     ;  RSA_N     = (volatile int*)0x0450 ;
/* RSA N modulus */
volatile int *RSA_K     ;  RSA_K     = (volatile int*)0x0490 ;
/* RSA K constant */
volatile int *RSA_len   ;  RSA_len   = (volatile int*)0x04D0 ;
/* RSA total message length as 32-bit address */
volatile int *RSA_conf  ;  RSA_conf  = (volatile int*)0x04E0 ;
/* RSA configuration register - RSA mode (mode=(00)-512/mode=(01)-
1024/mode=(10)-2048) */
volatile int *RSA_ED    ;  RSA_ED    = (volatile int*)0x04F0 ;
/* RSA encryption/decryption select */
volatile int *MSG       ;  MSG       = (volatile int*)0x0500 ;
```

```
/* message */
volatile int *CCM_U     ;  CCM_U     = (volatile int*)0x0E00 ;
/* CCM output's U */
volatile int *ZPU_RDY   ;  ZPU_RDY   = (volatile int*)0x0F00 ;
/* ZPU ready register */
volatile int *ZPU_CSR   ;  ZPU_CSR   = (volatile int*)0x0FFF ;
/* ZPU command/status register */

volatile int *AES_in    ;  AES_in    = (volatile int*)0x1000 ;
/* AES input */
volatile int *AES_out   ;  AES_out   = (volatile int*)0x1100 ;
/* AES output */
volatile int *AES_key   ;  AES_key   = (volatile int*)0x1200 ;
/* AES key */
volatile int *AES_mod   ;  AES_mod   = (volatile int*)0x1300 ;
/* AES mode */
volatile int *AES_CSR   ;  AES_CSR   = (volatile int*)0x1F00 ;
/* AES command/status register */

volatile int *SHA_in    ;  SHA_in    = (volatile int*)0x2000 ;
/* SHA input */
volatile int *SHA_out   ;  SHA_out   = (volatile int*)0x2100 ;
/* SHA output */
volatile int *SHA_clr   ;  SHA_clr   = (volatile int*)0x2200 ;
/* SHA clear */
volatile int *SHA_CSR   ;  SHA_CSR   = (volatile int*)0x2F00 ;
/* SHA command/status register */

volatile int *MMM_Ain   ;  MMM_Ain   = (volatile int*)0x3000 ;
/* MMM A_input */
volatile int *MMM_Bin   ;  MMM_Bin   = (volatile int*)0x3100 ;
/* MMM B_input */
volatile int *MMM_Cin   ;  MMM_Cin   = (volatile int*)0x3200 ;
/* MMM C_input */
volatile int *MMM_Yout  ;  MMM_Yout  = (volatile int*)0x3300 ;
/* MMM Y_output */
volatile int *MMM_mod   ;  MMM_mod   = (volatile int*)0x3400 ;
/* MMM mode */
volatile int *MMM_CSR   ;  MMM_CSR   = (volatile int*)0x3F00 ;
/* MMM command/status register */
```

```
/*
 *
 * aes_ccm_ipsec_esp.c
 * C code for AES-CCM
 *
 */

#include <stdio.h>
#include <stdlib.h>


void aes_ccm_ipsec_esp() {

    #include <ipsec.h>
    unsigned i , j  ;
    unsigned cnt    ;
    unsigned rpt    ;
    unsigned wpt    ;
```

```
    unsigned tmp    ;
    unsigned mt1[4] ;
    unsigned mt2[4] ;


    /* AES mode assignment */
    *AES_mod = *CCM_conf ;

    /* AES key assignment */
    AES_key[0] = CCM_Kd[0] ;
    AES_key[1] = CCM_Kd[1] ;
    AES_key[2] = CCM_Kd[2] ;
    AES_key[3] = CCM_Kd[3] ;

    if  (*CCM_conf>0) {
            AES_key[4] = CCM_Kd[4] ;
            AES_key[5] = CCM_Kd[5] ;
    }
    if  (*CCM_conf>1) {
            AES_key[6] = CCM_Kd[6] ;
            AES_key[7] = CCM_Kd[7] ;
    }
    /* End of AES key assignment */

    /* AES data assignment and encryption */
    /* B0 */
    AES_in[0] = (0<<31) | (1<<30) | (((*CCM_M-2)/2) << 27)  |
                (3<<24) | ((*CCM_NNCs<<8)>>8) ;
    AES_in[1] = CCM_NNCiv[0] ;
    AES_in[2] = CCM_NNCiv[1] ;
    AES_in[3] = *CCM_lm ;

    /* Set start = 1 */
    *AES_CSR = 1 ;
    /* Stalls ZPU - AES core active */
    /* When process finished, program jumps to the next line */

    /* B1 */
    if      ( *CCM_la == 8  ) {
        tmp    = (CCM_AAD[0] >> 16) & 0xFFFF     ;
        mt1[0] = (*CCM_la    << 16) | tmp        ;
        tmp    = (CCM_AAD[1] >> 16) & 0xFFFF     ;
        mt1[1] = (CCM_AAD[0] << 16) | tmp        ;
        tmp    = (CCM_AAD[1] << 16) & 0xFFFF0000 ;
        mt1[2] = tmp                             ;
        mt1[3] = 0                               ;
    }
    else if ( *CCM_la == 12 ) {
        tmp    = (CCM_AAD[0] >> 16) & 0xFFFF     ;
        mt1[0] = (*CCM_la    << 16) | tmp        ;
        tmp    = (CCM_AAD[1] >> 16) & 0xFFFF     ;
        mt1[1] = (CCM_AAD[0] << 16) | tmp        ;
        tmp    = (CCM_AAD[2] >> 16) & 0xFFFF     ;
        mt1[2] = (CCM_AAD[1] << 16) | tmp        ;
        tmp    = (CCM_AAD[2] << 16) & 0xFFFF0000 ;
        mt1[3] = tmp                             ;
    }

    AES_in[0] = mt1[0] ^ AES_out[0];
    AES_in[1] = mt1[1] ^ AES_out[1];
    AES_in[2] = mt1[2] ^ AES_out[2];
```

```
    AES_in[3] = mt1[3] ^ AES_out[3];

    /* Set start = 1 */
    *AES_CSR = 1 ;
    /* Stalls ZPU - AES core active */
    /* When process finished, program jumps to the next line */

    /* */
    wpt = 0   ;
    cnt = 1   ;
    for (i=0; i<(*CCM_lm>>2); i++) {

        /* B2 ... B(K+P) */
        mt1[wpt] = MSG[i] ;
        wpt++ ;

        if (wpt == 4) {
            mt2[0] = mt1[0] ^ AES_out[0];
            mt2[1] = mt1[1] ^ AES_out[1];
            mt2[2] = mt1[2] ^ AES_out[2];
            mt2[3] = mt1[3] ^ AES_out[3];
        }

        /* A1 ... A(R) */
        AES_in[0] = (0<<31) | (0<<30) | (0<<29) | (0<<28) |
                    (0<<27) | (3<<24) | ((*CCM_NNCs<<8)>>8) ;
        AES_in[1] = CCM_NNCiv[0] ;
        AES_in[2] = CCM_NNCiv[1] ;
        AES_in[3] = cnt          ;

        /* Set start = 1 */
        *AES_CSR = 1 ;
        /* Stalls ZPU - AES core active */
        /* When process finished, program jumps to the next line
*/

        MSG[0+(4*(cnt-1))] = mt2[0] ^ AES_out[0] ;
        MSG[1+(4*(cnt-1))] = mt2[1] ^ AES_out[1] ;
        MSG[2+(4*(cnt-1))] = mt2[2] ^ AES_out[2] ;
        MSG[3+(4*(cnt-1))] = mt2[3] ^ AES_out[3] ;

        AES_in[0] = mt2[0] ;
        AES_in[1] = mt2[1] ;
        AES_in[2] = mt2[2] ;
        AES_in[3] = mt2[3] ;

        /* Set start = 1 */
        *AES_CSR = 1 ;
        /* Stalls ZPU - AES core active */
        /* When process finished, program jumps to the next line
*/

        wpt = 0 ;
        cnt++   ;
    }

    if (((*CCM_lm<<28)>>30)!=0) {

        /* B2 ... B(K+P) */
        for (i=0; i<(4-((*CCM_lm<<28)>>30)); i++) {
            mt1[wpt] = 0 ; wpt++ ;
```

```
        }

        mt2[0] = mt1[0] ^ AES_out[0] ;
        mt2[1] = mt1[1] ^ AES_out[1] ;
        mt2[2] = mt1[2] ^ AES_out[2] ;
        mt2[3] = mt1[3] ^ AES_out[3] ;

        /* A1 ... A(R) */
        AES_in[0] = (0<<31) | (0<<30) | (0<<29) | (0<<28) |
                    (0<<27) | (3<<24) | ((*CCM_NNCs<<8)>>8) ;
        AES_in[1] = CCM_NNCiv[0] ;
        AES_in[2] = CCM_NNCiv[1] ;
        AES_in[3] = cnt          ;

        /* Set start = 1 */
        *AES_CSR = 1 ;
        /* Stalls ZPU - AES core active */
        /* When process finished, program jumps to the next line
*/

        MSG[0+(4*(cnt-1))] = mt2[0] ^ AES_out[0] ;
        MSG[1+(4*(cnt-1))] = mt2[1] ^ AES_out[1] ;
        MSG[2+(4*(cnt-1))] = mt2[2] ^ AES_out[2] ;
        MSG[3+(4*(cnt-1))] = mt2[3] ^ AES_out[3] ;

        AES_in[0] = mt2[0] ;
        AES_in[1] = mt2[1] ;
        AES_in[2] = mt2[2] ;
        AES_in[3] = mt2[3] ;

        /* Set start = 1 */
        *AES_CSR = 1 ;
        /* Stalls ZPU - AES core active */
        /* When process finished, program jumps to the next line
*/
    }

    /* A0 */
    mt2[0] = AES_out[0] ;
    mt2[1] = AES_out[1] ;
    mt2[2] = AES_out[2] ;
    mt2[3] = AES_out[3] ;

    AES_in[0] = (0<<31) | (0<<30) | (0<<29) | (0<<28) | (0<<27) |
                (3<<24) | ((*CCM_NNCs<<8)>>8) ;
    AES_in[1] = CCM_NNCiv[0] ;
    AES_in[2] = CCM_NNCiv[1] ;
    AES_in[3] = 0            ;

    /* Set start = 1 */
    *AES_CSR = 1 ;
    /* Stalls ZPU - AES core active */
    /* When process finished, program jumps to the next line */

    if (*CCM_M<12) {
        CCM_U[0] = mt2[0] ^ AES_out[0] ;
        CCM_U[1] = mt2[1] ^ AES_out[1] ;
    }
    else if (*CCM_M<16) {
        CCM_U[2] = mt2[2] ^ AES_out[2] ;
    }
```

```
    else
        CCM_U[3] = mt2[3] ^ AES_out[3] ;
    /* End of AES data assignment and encryption */

}



int main(void) {

    #include <ipsec.h>
    unsigned i ;


    *CCM_M  = 16 ; /* CCM ICV length     */

    *CCM_lm = 36 ; /* CCM message length */

    *CCM_la = 12 ; /* CCM AAD length     */

    *CCM_NNCs    = 0x00111213 ; /* salt --- CCM nonce */
    CCM_NNCiv[0] = 0x21222324 ; /* IV   --- CCM nonce */
    CCM_NNCiv[1] = 0x31323334 ; /* IV   --- CCM nonce */

    CCM_AAD[0] = 0xB0B1B2B3 ; /* CCM AAD */
    CCM_AAD[1] = 0xB4B5B6B7 ; /* CCM AAD */
    CCM_AAD[2] = 0xB8B9BABB ; /* CCM AAD */

    for (i=0; i<8; i++) {
        CCM_Kd[i] = i ; /* CCM key input */
    }

    *CCM_conf = 0 ; /* CCM configuration register - Key mode
(mode=(00)-128/mode=(01)-192/mode=(10)-256) */

    MSG[0]  = 0xC0C1C2C3 ;  /* message */
    MSG[1]  = 0xC4C5C6C7 ;  /* message */
    MSG[2]  = 0xC8C9D0D1 ;  /* message */
    MSG[3]  = 0xD2D3D4D5 ;  /* message */
    MSG[4]  = 0xD6D7D8D9 ;  /* message */
    MSG[5]  = 0xE0E1E2E3 ;  /* message */
    MSG[6]  = 0xE4E5E6E7 ;  /* message */
    MSG[7]  = 0xF0F1F2F3 ;  /* message */
    MSG[8]  = 0xF4F5F6F7 ;  /* message */
    MSG[9]  = 0x11223344 ;  /* message */
    MSG[10] = 0x55667788 ;  /* message */
    MSG[11] = 0x99AABBCC ;  /* message */
    MSG[12] = 0xDDEEFF00 ;  /* message */
    MSG[13] = 0x12345678 ;  /* message */
    MSG[14] = 0x90ABCDEF ;  /* message */
    MSG[15] = 0xAABBCCDD ;  /* message */

    CCM_U[0] = 0 ;  /* CCM output's U */
    CCM_U[1] = 0 ;
    CCM_U[2] = 0 ;
    CCM_U[3] = 0 ;

    AES_in[0] = 0 ;  /* AES input */
    AES_in[1] = 0 ;
    AES_in[2] = 0 ;
    AES_in[3] = 0 ;
```

```
    AES_key[0] = 0 ;  /* AES key */
    AES_key[1] = 0 ;
    AES_key[2] = 0 ;
    AES_key[3] = 0 ;
    AES_key[4] = 0 ;
    AES_key[5] = 0 ;
    AES_key[6] = 0 ;
    AES_key[7] = 0 ;

    *AES_mod = 0 ;  /* AES mode */

    aes_ccm_ipsec_esp() ;

}
```

```
/*
 *
 * sha_hmac_ipsec_esp.c
 * C code for HMAC-SHA-1-96
 *
 */

#include <stdio.h>
#include <stdlib.h>


void sha_hmac_ipsec_esp(void) {

      #include <ipsec.h>
    unsigned i ;
    unsigned done = 0 ;
    unsigned cnt  = 0 ;
    unsigned mt[5] ;

      /**** B0 ****/
    for (i=0; i<5; i++) {
          SHA_in[i] = HMAC_Kd[i] ^ 0x36363636 ;
      }
    for (i=5; i<16; i++) {
          SHA_in[i] = 0x36363636 ;
      }

      /* Set clear = 1 */
      *SHA_clr = 1 ;
      /* For first input block */

      /* Set start = 1 */
      *SHA_CSR = 1 ;
      /* Stalls ZPU - SHA core active */
      /* When process finished, program jumps to the next line */
      /**** B0 ****/

      /**** B1 ... B(N) ****/
    if ((HMAC_lm[0]!=0)||(HMAC_lm[1]!=0)) {

            while (done==0) {

                  for (i=0; i<16; i++) {
```

```
                        SHA_in[i] = MSG[i+(cnt*16)] ;
                  }

                  if ((HMAC_lm[0]!=0)&&(HMAC_lm[1]!=0)) {
                        if (HMAC_lm[1]==0) {
                              HMAC_lm[0]-- ;
                              HMAC_lm[1] = 0xFFFFFFC0 ;
                        }
                        else {
                              HMAC_lm[1] = HMAC_lm[1]-64 ;
                        }
                        cnt++ ;
                  }
                  else {
                        HMAC_lm[1] = HMAC_lm[1]-64 ;
                        if (HMAC_lm[1]==0) {
                              done = 1 ;
                        }
                        cnt++ ;
                  }

                  /* Set start = 1 */
                  *SHA_CSR = 1 ;
                  /* Stalls ZPU - SHA core active */
                  /* When process finished, program jumps to the
next line */

            }

      }
      /**** B1 ... B(N) ****/

      for (i=0; i<5; i++) {
            mt[i] = SHA_out[i] ;
      }


      /**** A0 ****/
    for (i=0; i<5; i++) {
          SHA_in[i] = HMAC_Kd[i] ^ 0x5C5C5C5C ;
      }
    for (i=5; i<16; i++) {
         SHA_in[i] = 0x5C5C5C5C ;
      }

      /* Set clear = 1 */
      *SHA_clr = 1 ;
      /* For first input block */

      /* Set start = 1 */
      *SHA_CSR = 1 ;
      /* Stalls ZPU - SHA core active */
      /* When process finished, program jumps to the next line */
      /**** A0 ****/


      /**** A1 ****/
    for (i=0; i<5; i++) {
         SHA_in[i] = mt[i] ;
      }
      SHA_in[5] = 0x80000000 ;
```

```
    for (i=6; i<15; i++) {
         SHA_in[i] = 0 ;
      }
    SHA_in[15] = 0x000002A0 ;

      /* Set start = 1 */
      *SHA_CSR = 1 ;
      /* Stalls ZPU - SHA core active */
      /* When process finished, program jumps to the next line */
      /**** A1 ****/


}



int main(void) {

    #include <ipsec.h>
    unsigned i ;

    HMAC_lm[0] = 0x00000000 ;  /* HMAC message length */
    HMAC_lm[1] = 0x00000040 ;  /* HMAC message length */

    HMAC_Kd[0] = 0x12345678 ;  /* HMAC key input */
    HMAC_Kd[1] = 0x90ABCDEF ;  /* HMAC key input */
    HMAC_Kd[2] = 0x11223344 ;  /* HMAC key input */
    HMAC_Kd[3] = 0x55667788 ;  /* HMAC key input */
    HMAC_Kd[4] = 0x9900AABB ;  /* HMAC key input */

    MSG[0]  = 0xC0C1C2C3 ;     /* message */
    MSG[1]  = 0xC4C5C6C7 ;     /* message */
    MSG[2]  = 0xC8C9D0D1 ;     /* message */
    MSG[3]  = 0xD2D3D4D5 ;     /* message */
    MSG[4]  = 0xD6D7D8D9 ;     /* message */
    MSG[5]  = 0xE0E1E2E3 ;     /* message */
    MSG[6]  = 0xE4E5E6E7 ;     /* message */
    MSG[7]  = 0xF0F1F2F3 ;     /* message */
    MSG[8]  = 0xF4F5F6F7 ;     /* message */
    MSG[9]  = 0x11223344 ;     /* message */
    MSG[10] = 0x55667788 ;     /* message */
    MSG[11] = 0x99AABBCC ;     /* message */
    MSG[12] = 0xDDEEFF00 ;     /* message */
    MSG[13] = 0x12345678 ;     /* message */
    MSG[14] = 0x90ABCDEF ;     /* message */
    MSG[15] = 0xAABBCCDD ;     /* message */

      for (i=0; i<16; i++) {
           SHA_in[i] = 0 ;        /* SHA input */
      }

      for (i=0; i<5; i++) {
           SHA_out[i] = 0 ;       /* SHA output */
      }

      sha_hmac_ipsec_esp() ;

}
```

```
/*
 *
 * mmm_rsa_ipsec_esp.c
 * C code for RSA
 *
 */

#include <stdio.h>
#include <stdlib.h>


void mmm_rsa_ipsec_esp(void) {

    #include <ipsec.h>
    unsigned i    , j     ;
    unsigned len  , sel   ;
    unsigned cnt1 , cnt2  ;
    unsigned e    , e_bit ;
    unsigned m[64] ;
    unsigned r[64] ;


    /* RSA mode select */
    if      ( *RSA_conf == 0 ) {
        sel = 16 ;
        len = (*RSA_len>>4) ;
    }
    else if ( *RSA_conf == 1 ) {
        sel = 32 ;
        len = (*RSA_len>>5) ;
    }
    else {
        sel = 64 ;
        len = (*RSA_len>>6) ;
    }

    /* MMM mode assignment */
    *MMM_mod = *RSA_conf ;
    /* End of AES mode assignment */

    for (i=0; i<len; i++) {

        for (j=0; j<sel; j++) {
            m[j] = MSG[(sel*i)+j] ;
        }

        /*** MME calculation ***/

        /* m = MMM(m,k,n) */
        /* Set MMM's input C */
        for (j=0; j<sel; j++) {
            MMM_Cin[j] = RSA_N[j] ;
        }
        /* Set MMM's input A */
        for (j=0; j<sel; j++) {
            MMM_Ain[j] = m[j] ;
        }
        /* Set MMM's input B */
        for (j=0; j<sel; j++) {
            MMM_Bin[j] = RSA_K[j] ;
        }
```

```
        /**/
        /* Set start = 1 */
        *MMM_CSR = 1 ;
        /* Stalls ZPU - MMM core active */
        /* When process finished, program jumps to the next line
*/
        /**/

        /* Set m to MMM's output Y */
        for (j=0; j<sel; j++) {
            m[j] = MMM_Yout[j] ;
        }

        /* r = MMM(1,k,n) */
        /* Set MMM's input A */
        for (j=0; j<(sel-1); j++) {
            MMM_Ain[j] = 0 ;
        }
        MMM_Ain[sel-1] = 1 ;

        /**/
        /* Set start = 1 */
        *MMM_CSR = 1 ;
        /* Stalls ZPU - MMM core active */
        /* When process finished, program jumps to the next line
*/
        /**/

        /* Set r to MMM's output Y */
        for (j=0; j<sel; j++) {
            r[j] = MMM_Yout[j] ;
        }

        /** For loop of MME **/
        if (*RSA_ED!=0) {
            cnt1 = 1 ;
        }
        else {
            cnt1 = sel ;
        }

        while (cnt1 != 0) {

            cnt2 = 32 ;

            if   (*RSA_ED!=0) {
                e = *RSA_e ;
            }
            else {
                e = RSA_d[cnt1-1] ;
            }

            while (cnt2!=0) {

                e_bit = (e<<31) ;
                e = (e>>1) ;
                cnt2-- ;

                /* RSA operations of the for loop */
                /* Set MMM's input B */
```

```
                for (j=0; j<sel; j++) {
                    MMM_Bin[j] = m[j] ;
                }

                /**/
                if (e_bit!=0) {

                    /* r = MMM(r,m,n) */
                    /* Set MMM's input A */
                    for (j=0; j<sel; j++) {
                        MMM_Ain[j] = r[j] ;
                    }

                    /**/
                    /* Set start = 1 */
                    *MMM_CSR = 1 ;
                    /* Stalls ZPU - MMM core active */
                    /* When process finished, program jumps to the
next line */
                    /**/

                    /* Set r to MMM's output Y */
                    for (j=0; j<sel; j++) {
                        r[j] = MMM_Yout[j] ;
                    }

                }

                /* m = MMM(m,m,n) */
                /* Set MMM's input A */
                for (j=0; j<sel; j++) {
                    MMM_Ain[j] = m[j] ;
                }

                /**/
                /* Set start = 1 */
                *MMM_CSR = 1 ;
                /* Stalls ZPU - MMM core active */
                /* When process finished, program jumps to the
next line */
                /**/

                /* Set m to MMM's output Y */
                for (j=0; j<sel; j++) {
                    m[j] = MMM_Yout[j] ;
                }
                /* End of RSA operations of the for loop */
            }

            cnt1-- ;

        }
        /** End of for loop of MME **/

        /* r = MMM(r,1,n) */
        /* Set MMM's input A */
        for (j=0; j<sel; j++) {
            MMM_Ain[j] = r[j] ;
        }
        /* Set MMM's input B */
        for (j=0; j<(sel-1); j++) {
```

```
            MMM_Bin[j] = 0 ;
        }
        MMM_Bin[sel-1] = 1 ;

        /**/
        /* Set start = 1 */
        *MMM_CSR = 1 ;
        /* Stalls ZPU - MMM core active */
        /* When process finished, program jumps to the next line
*/
        /**/

        /* Set r to MMM's output Y */
        for (j=0; j<sel; j++) {
            r[j] = MMM_Yout[j] ;
        }

        /*** End of MME calculation ***/

        for (j=0; j<sel; j++) {
            MSG[(sel*i)+j] = r[j] ;
        }

    }

}



int main(void) {

    #include <ipsec.h>
    unsigned i ;


    for (i=0; i<64; i++) {
        RSA_d[i] = i ;      /* rsa d private key */
    }

    *RSA_e = 0x00010001 ;   /* rsa e public key */

    for (i=0; i<64; i++) {
        RSA_N[i] = i ;      /* rsa N modulus */
    }

    for (i=0; i<64; i++) {
        RSA_K[i] = i ;      /* rsa K constant */
    }

    *RSA_len = 16 ;      /* rsa total message length as 32-bit
address*/

    *RSA_conf = 0 ;      /* rsa configuration register */

    *RSA_ED   = 1 ;      /* rsa encryption or decryption select
register*/

    MSG[0]  = 0xC0C1C2C3 ;  /* message */
    MSG[1]  = 0xC4C5C6C7 ;  /* message */
    MSG[2]  = 0xC8C9D0D1 ;  /* message */
    MSG[3]  = 0xD2D3D4D5 ;  /* message */
```

```
    MSG[4]  = 0xD6D7D8D9 ;  /* message */
    MSG[5]  = 0xE0E1E2E3 ;  /* message */
    MSG[6]  = 0xE4E5E6E7 ;  /* message */
    MSG[7]  = 0xF0F1F2F3 ;  /* message */
    MSG[8]  = 0xF4F5F6F7 ;  /* message */
    MSG[9]  = 0x11223344 ;  /* message */
    MSG[10] = 0x55667788 ;  /* message */
    MSG[11] = 0x99AABBCC ;  /* message */
    MSG[12] = 0xDDEEFF00 ;  /* message */
    MSG[13] = 0x12345678 ;  /* message */
    MSG[14] = 0x90ABCDEF ;  /* message */
    MSG[15] = 0xAABBCCDD ;  /* message */

    *MMM_mod = 0 ;          /* MMM mode */

    for (i=0; i<64; i++) {
        MMM_Ain[i] = 0 ;    /* MMM A input */
        MMM_Bin[i] = 0 ;    /* MMM B input */
        MMM_Cin[i] = 0 ;    /* MMM C input */
        MMM_Yout[i] = 0 ;   /* MMM Y output */
    }

    mmm_rsa_ipsec_esp() ;

}
```