NATIONAL TECHNICAL UNIVERSITY OF ATHENS
SCHOOL OF ELECTRICAL AND COMPUTER ENGINEERING
DIVISION OF COMPUTER SCIENCE

Design and Implementation of a Versatile Hardware Crypto IP for Symmetric and Asymmetric Algorithms

DIPLOMA THESIS

NIKOLAOS A. EFTAXIOPOULOS - SARRIS

GEORGIOS D. ZERVAKIS

Supervisor: Kiamal Z. Pekmestzi

Professor

Athens, October 2012

ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ
ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ
ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ
ΤΟΜΕΑΣ ΤΕΧΝΟΛΟΓΙΑΣ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΥΠΟΛΟΓΙΣΤΩΝ

Σχεδίαση και Υλοποίηση Κυκλώματος επί Ψηφίδας για Αλγόριθμους Κρυπτογραφίας Συμμετρικού και Ασύμμετρου Κλειδιού

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

ΝΙΚΟΛΑΟΣ Α. ΕΥΤΑΞΙΟΠΟΥΛΟΣ - ΣΑΡΡΗΣ

ΓΕΩΡΓΙΟΣ Δ. ΖΕΡΒΑΚΗΣ

Επιβλέπων: Κιαμάλ Ζ. Πεκμεστζή

Καθηγητής

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την 30η Οκτωβρίου 2012.

.....	.....	.....
Κιαμάλ Πεκμεστζή	Δημήτριος Σούντρης	Γεώργιος Οικονομάκος
Καθηγητής	Επίκουρος Καθηγητής	Λέκτορας

Αθήνα, Οκτώβριος 2012

.....

ΝΙΚΟΛΑΟΣ Α. ΕΥΤΑΞΙΟΠΟΥΛΟΣ – ΣΑΡΡΗΣ
ΓΕΩΡΓΙΟΣ Δ. ΖΕΡΒΑΚΗΣ

Διπλωματούχοι Ηλεκτρολόγοι Μηχανικοί και Μηχανικοί Υπολογιστών Ε.Μ.Π.

Table of Contents

Abstract

This diploma thesis was developed in the Microprocessors and Digital Systems Lab of National Technical University of Athens. As we attended the courses of this lab like Microprocessors Systems, Microprocessors Laboratory and Digital VLSI Systems a great interest was developed in the area of hardware description languages and specifically in the implementation of hardware circuits since in this area, theory and practice converge. That interest became more intense under the observation that the implementation of these circuits is the core of the semiconductor and embedded systems industry with a vast amount of applications in the daily life. An intriguing category of applications is the one related to cryptography. This is the reason why this diploma thesis focuses in the implementation of certain symmetric-key and public-key cryptographic algorithms like DES (Data Encryption Standard), AES (Advanced Encryption Standard), IDEA (International Data Encryption Algorithm) and RSA. As a result a cryptographic IP was designed, implemented and evaluated, called Crypto IP.

This thesis includes six chapters. Chapter 1 contains a brief introduction in cryptography, a detailed description of the implemented symmetric-key and public-key algorithms, as well as references in each one's applications in the daily life. In the second chapter the overall architecture of the Crypto IP is described: the main circuits implementing the cryptographic algorithms and a communication interface with the widely used AMBA bus so that the IP can be connected to a System on Chip. In chapters 3, 4 and 5 there is a detailed description of the implementation of each cryptographic circuit as well as a reference to the parameters which can be configured by the user. In chapter 6 there is a brief reference in the way that the functionality of these circuits was verified on an FPGA board using an external controller which feeds the circuits with the proper test cases. This thesis ends with an Appendix where the basic tables used by DES and AES algorithm are mentioned.

Finally we would like to thank the professor and laboratory supervisor Mr. K. Pekmestzi, the lecturer Mr. N. Moshopoulos whose experience led as in the production of a completed project according to the industry standards and guided us during the entire period of the thesis development, as well as the PhD students K. Tsoumanis and D. Bekiaris for the continuous technical support provided and their valuable advices.

Key Words

- Integrated circuits
- Cryptography
- Crypto IP
- Symmetric key
- Asymmetric key
- Public key
- Private key
- DES
- AES
- IDEA
- RSA
- AMBA AHB
- FPGA
- ASIC

Περίληψη

Η διπλωματική αυτή εργασία εκπονήθηκε στο εργαστήριο Μικροϋπολογιστών και Ψηφιακών Συστημάτων του Εθνικού Μετσόβιου Πολυτεχνείου. Έχοντας παρακολουθήσει τα μαθήματα του συγκεκριμένου εργαστηρίου όπως τα Συστήματα Μικροϋπολογιστών, το Εργαστήριο Μικροϋπολογιστών και τα Ψηφιακά Συστήματα VLSI αναπτύχθηκε ένα ιδιαίτερο ενδιαφέρον στον τομέα των γλωσσών περιγραφής υλικού και συγκεκριμένα στην υλοποίηση κυκλωμάτων σε επίπεδο hardware καθώς αποτελεί έναν τομέα όπου η θεωρία με την πράξη είναι αλληλένδετες. Το ενδιαφέρον έγινε ακόμα πιο έντονο παρατηρώντας ότι η υλοποίηση κυκλωμάτων αποτελεί τον πυρήνα της βιομηχανίας ημιαγωγών και ενσωματωμένων συστημάτων με πληθώρα εφαρμογών στην καθημερινή ζωή. Μια ιδιαίτερα ενδιαφέρουσα κατηγορία εφαρμογών είναι αυτή της κρυπτογραφίας. Για τον λόγο αυτό αποφασίστηκε η συγκεκριμένη διπλωματική να επικεντρωθεί στην υλοποίηση κάποιων βασικών αλγόριθμων κρυπτογραφίας συμμετρικού και ασύμμετρου (δημόσιου) κλειδιού όπως ο DES (Data Encryption Standard), ο AES (Advanced Encryption Standard), ο IDEA (International Data Encryption Algorithm) και ο RSA. Αποτέλεσμα ήταν η σχεδίαση, υλοποίηση και αξιολόγηση ενός IP κρυπτογραφίας με την ονομασία Crypto IP.

Η εργασία αυτή περιλαμβάνει έξι κεφάλαια. Το κεφάλαιο 1 περιέχει μία σύντομη εισαγωγή στην κρυπτογραφία, αναλυτική περιγραφή των αλγορίθμων συμμετρικού και δημόσιου κλειδιού που υλοποιήθηκαν, καθώς και μια αναφορά στις εφαρμογές του καθενός στην καθημερινή ζωή. Στο δεύτερο κεφάλαιο περιγράφεται η συνολική αρχιτεκτονική του Crypto IP: τα κύρια κυκλώματα που υλοποιούν του αλγόριθμους και μία διεπαφή για επικοινωνία με τον ευρέως χρησιμοποιούμενο διάδρομο AMBA ώστε το Crypto IP να έχει τη δυνατότητα διασύνδεσης σε ένα Σύστημα επί Ψηφίδας. Στα κεφάλαια 3, 4 και 5 γίνεται αναλυτική περιγραφή της υλοποίησης του κάθε κρυπτογραφικού κυκλώματος με αναφορά στις δυνατότητες παραμετροποίησης του από τον χρήστη. Στο κεφάλαιο 6 γίνεται μία σύντομη αναφορά στον τρόπο επαλήθευσης της ορθής λειτουργίας των κυκλωμάτων αυτών σε μια πλακέτα FPGA με τη χρήση ενός εξωτερικού ελεγκτή που τα τροφοδοτεί με τις κατάλληλες περιπτώσεις ελέγχου. Η εργασία ολοκληρώνεται με ένα παράρτημα στο οποίο παραθέτονται κάποιοι βασικοί πίνακες που χρησιμοποιούνται στους αλγορίθμους DES και AES.

Τέλος θα θέλαμε να ευχαριστήσουμε τον επιβλέποντα καθηγητή και υπεύθυνο του εργαστηρίου κύριο Κ. Πεκμεστζή, τον λέκτορα κύριο Ν. Μοσχόπουλο που μέσω της εμπειρίας του μας οδήγησε στην παραγωγή ενός ολοκληρωμένου έργου στα πρότυπα της

βιομηχανίας και μας καθοδηγούσε σε όλη τη διάρκεια εκπόνησης της εργασίας, καθώς και τους διδακτορικούς φοιτητές Κ. Τσουμάνη και Δ. Μπεκιάρη για την συνεχή τεχνική υποστήριξη που μας παρείχαν και τις πολύτιμες συμβουλές τους.

Λέξεις - Κλειδιά

- Ολοκληρωμένα κυκλώματα
- Κρυπτογραφία
- Crypto IP
- Συμμετρικό κλειδί
- Ασύμμετρο κλειδί
- Δημόσιο κλειδί
- Ιδιωτικό κλειδί
- DES
- AES
- IDEA
- RSA
- AMBA AHB
- FPGA
- ASIC

1 Cryptography

1.1 Introduction

Cryptography comes from the Greek words κρύπτος (hidden, secret) and γράφειν (writing) and is the practice and study of techniques for secure communication in the presence of third parties (called adversaries). More generally, it is about constructing and analyzing protocols that overcome the influence of adversaries and which are related to various aspects in information security such as data confidentiality, data integrity, authentication, and non-repudiation.

Cryptography, prior to the modern age, was effectively synonymous with encryption, the conversion of information from a readable state to apparent nonsense. The originator of an encrypted message shared the decoding technique needed to recover the original information only with intended recipients, thereby precluding unwanted persons to do the same. Since World War I and the advent of the computer, the methods used to carry out cryptology have become increasingly complex and its application more widespread. Encryption was used to (attempt to) ensure secrecy in communications, such as those of spies, military leaders, and diplomats. In recent decades, the field has expanded beyond confidentiality concerns to include techniques for message integrity checking, sender/receiver identity authentication, digital signatures, interactive proofs and secure computation, among others.

Modern cryptography is heavily based on mathematical theory and computer science practice. Cryptographic algorithms are designed around computational hardness assumptions, making such algorithms hard to break in practice by any adversary. It is theoretically possible to break such a system but it is infeasible to do so by any known practical means. These schemes are therefore termed computationally secure. Theoretical advances (e.g. improvements in integer factorization algorithms) and faster computing technology require these solutions to be continually adapted. There exist information-theoretically secure schemes that provably cannot be broken even with unlimited computing power (an example is the one-time pad) but these schemes are more difficult to implement than the best theoretically breakable but computationally secure mechanisms.

The modern field of cryptography can be divided into several areas of study. The chief ones are the symmetric-key cryptography and the public-key cryptography.

1.1.1 Symmetric-key Cryptography

Symmetric-key cryptography refers to encryption methods in which both the sender and receiver share the same key (or, less commonly, in which their keys are different, but related in an easily computable way). This was the only kind of encryption publicly known until June 1976. Symmetric key ciphers are implemented as either block ciphers or stream ciphers. A block cipher enciphers input in blocks of plaintext as opposed to individual characters, the input form used by a stream cipher.

The Data Encryption Standard (DES) and the Advanced Encryption Standard (AES) are block cipher designs which have been designated cryptography standards by the US government (though DES's designation was finally withdrawn after the AES was adopted). Despite its deprecation as an official standard, DES (especially its still-approved and much more secure triple-DES variant) remains quite popular and it is used across a wide range of applications.

Stream ciphers, in contrast to the 'block' type, create an arbitrarily long stream of key material, which is combined with the plaintext bit-by-bit or character-by-character, somewhat like the one-time pad. In a stream cipher, the output stream is created based on a hidden internal state which changes as the cipher operates. That internal state is initially set up using the secret key material. Block ciphers can be used as stream ciphers using certain modes of operation.

Cryptographic hash functions are a third type of cryptographic algorithm. They take a message of any length as input, and output a short, fixed length hash which can be used in (for example) a digital signature. For good hash functions, an attacker cannot find two messages that produce the same hash. MD4 is a long-used hash function which is now broken. MD5, a strengthened variant of MD4, is also widely used but broken in practice. The U.S. National Security Agency (NSA[1]) developed the Secure Hash Algorithm series of MD5-

[1] The National Security Agency (NSA) is a cryptologic intelligence agency of the United States Department of Defense. (http://www.nsa.gov)

like hash functions. SHA-0 was a flawed algorithm that the agency withdrew. SHA-1 is widely deployed and more secure than MD5, but cryptanalysts have identified attacks against it. The SHA-2 family improves on SHA-1, but it isn't yet widely deployed, and the U.S. standards authority thought it "prudent" from a security perspective to develop a new standard to significantly improve the robustness of NIST's[2] overall hash algorithm toolkit. Message authentication codes (MACs) are much like cryptographic hash functions, except that a secret key can be used to authenticate the hash value upon receipt.

1.1.2 Public-key Cryptography

Symmetric-key cryptosystems use the same key for encryption and decryption of a message, though a message or group of messages may have a different key than others. A significant disadvantage of symmetric ciphers is the key management necessary to use them securely. Each distinct pair of communicating parties must, ideally, share a different key, and perhaps each ciphertext exchanged as well. The number of keys required increases as the square of the number of network members, which very quickly requires complex key management schemes to keep them all straight and secret. The difficulty of securely establishing a secret key between two communicating parties, when a secure channel does not already exist between them, also presents a chicken-and-egg problem which is a considerable practical obstacle for cryptography users in the real world.

In a groundbreaking 1976 paper, Whitfield Diffie and Martin Hellman proposed the notion of public-key (also, more generally, called asymmetric key) cryptography in which two different but mathematically related keys are used—a public key and a private key. A public key system is so constructed that calculation of one key (the 'private key') is computationally infeasible from the other (the 'public key'), even though they are necessarily related. Instead, both keys are generated secretly, as an interrelated pair. The historian David Kahn described public-key cryptography as "the most revolutionary new concept in the field since polyalphabetic substitution emerged in the Renaissance".

---

[2] The National Institute of Standards and Technology (NIST) is a measurement standards laboratory which is a non-regulatory agency of the United States Department of Commerce. (http://www.nist.gov)

In public-key cryptosystems, the public key may be freely distributed, while its paired private key must remain secret. In a public-key encryption system, the public key is used for encryption, while the private or secret key is used for decryption. While Diffie and Hellman could not find such a system, they showed that public-key cryptography was indeed possible by presenting the Diffie–Hellman key exchange protocol, a solution that is now widely used in secure communications to allow two parties to secretly agree on a shared encryption key.

Diffie and Hellman's publication sparked widespread academic efforts in finding a practical public-key encryption system. This race was finally won in 1978 by Ronald Rivest, Adi Shamir, and Len Adleman, whose solution has since become known as the RSA algorithm. The Diffie–Hellman and RSA algorithms, in addition to being the first publicly known examples of high quality public-key algorithms, have been among the most widely used. Others include the Cramer–Shoup cryptosystem, ElGamal encryption, and various elliptic curve techniques. Around 1970, James H. Ellis had conceived the principles of asymmetric key cryptography. In 1973, Clifford Cocks invented a solution that essentially resembles the RSA algorithm. And in 1974, Malcolm J. Williamson is claimed to have developed the Diffie-Hellman key exchange.

Public-key cryptography can also be used for implementing digital signature schemes. A digital signature is reminiscent of an ordinary signature. They both have the characteristic of being easy for a user to produce, but difficult for anyone else to forge. Digital signatures can also be permanently tied to the content of the message being signed. They cannot then be 'moved' from one document to another, for any attempt will be detectable. In digital signature schemes, there are two algorithms; one for signing, in which a secret key is used to process the message (or a hash of the message, or both), and one for verification, in which the matching public key is used with the message to check the validity of the signature. RSA and DSA are two of the most popular digital signature schemes. Digital signatures are central to the operation of public key infrastructures and many network security schemes (e.g. SSL/TLS, many VPNs etc.).

Public-key algorithms are most often based on the computational complexity of "hard" problems, often from number theory. For example, the hardness of RSA is related to the integer factorization problem, while Diffie–Hellman and DSA are related to the discrete logarithm problem. More recently, elliptic curve cryptography has developed in which security is based on number theoretic problems involving elliptic curves. Because of the difficulty of the underlying problems, most public-key algorithms involve operations such as modular multiplication and exponentiation, which are much more computationally

expensive than the techniques used in most block ciphers, especially with typical key sizes. As a result, public-key cryptosystems are commonly hybrid cryptosystems, in which a fast high-quality symmetric-key encryption algorithm is used for the message itself, while the relevant symmetric key is sent with the message, but encrypted using a public-key algorithm. Similarly, hybrid signature schemes are often used, in which a cryptographic hash function is computed, and only the resulting hash is digitally signed.

1.1.3 Cryptanalysis

The goal of cryptanalysis is to find some weakness or insecurity in a cryptographic scheme, thus permitting its subversion or evasion.

It is a common misconception that every encryption method can be broken. In connection with his WWII work at Bell Labs, Claude Shannon proved that the one-time pad cipher is unbreakable, provided the key material is truly random, never reused, kept secret from all possible attackers, and of equal or greater length than the message. Most ciphers, apart from the one-time pad, can be broken with enough computational effort by brute force attack, but the amount of effort needed may be exponentially dependent on the key size, as compared to the effort needed to make use of the cipher. In such cases, effective security could be achieved if it is proven that the effort required is beyond the ability of any adversary. This means it must be shown that no efficient method (as opposed to the time-consuming brute force method) can be found to break the cipher. Since no such proof has been found to date, the one-time-pad remains the only theoretically unbreakable cipher.

There are a wide variety of cryptanalytic attacks, and they can be classified in any of several ways. A common distinction turns on what an attacker knows and what capabilities are available. In a ciphertext-only attack, the cryptanalyst has access only to the ciphertext (good modern cryptosystems are usually effectively immune to ciphertext-only attacks). In a known-plaintext attack, the cryptanalyst has access to a ciphertext and its corresponding plaintext (or to many such pairs). In a chosen-plaintext attack, the cryptanalyst may choose a plaintext and learn its corresponding ciphertext (perhaps many times). Finally, in a chosen-ciphertext attack, the cryptanalyst may be able to choose ciphertexts and learn their corresponding plaintexts.

Cryptanalysis of symmetric-key ciphers typically involves looking for attacks against the block ciphers or stream ciphers that are more efficient than any attack that could be against a perfect cipher. For example, a simple brute force attack against DES requires one known plaintext and 255 decryptions, trying approximately half of the possible keys, to reach a point at which chances are better than even that the key sought will have been found. But this may not be enough assurance; a linear cryptanalysis attack against DES requires 243 known plaintexts and approximately 243 DES operations. This is a considerable improvement on brute force attacks.

Public-key algorithms are based on the computational difficulty of various problems. The most famous of these is integer factorization (e.g., the RSA algorithm is based on a problem related to integer factoring), but the discrete logarithm problem is also important. Much public-key cryptanalysis concerns numerical algorithms for solving these computational problems, or some of them, efficiently (i.e., in a practical time). For instance, the best known algorithms for solving the elliptic curve-based version of discrete logarithm are much more time-consuming than the best known algorithms for factoring, at least for problems of more or less equivalent size. Thus, other things being equal, to achieve an equivalent strength of attack resistance, factoring-based encryption techniques must use larger keys than elliptic curve techniques. For this reason, public-key cryptosystems based on elliptic curves have become popular since their invention in the mid-1990s.

While pure cryptanalysis uses weaknesses in the algorithms themselves, other attacks on cryptosystems are based on actual use of the algorithms in real devices, and are called side-channel attacks. If a cryptanalyst has access to, for example, the amount of time the device took to encrypt a number of plaintexts or report an error in a password or PIN character, he may be able to use a timing attack to break a cipher that is otherwise resistant to analysis. An attacker might also study the pattern and length of messages to derive valuable information; this is known as traffic analysis, and can be quite useful to an alert adversary. Poor administration of a cryptosystem, such as permitting too short keys, will make any system vulnerable, regardless of other virtues. And, of course, social engineering, and other attacks against the personnel who work with cryptosystems or the messages they handle may be the most productive attacks of all.

1.1.4 Legal Issues

Cryptology-related technology has raised a number of legal issues. In the United Kingdom, additions to the Regulation of Investigatory Powers Act 2000 require a suspected criminal to hand over their encryption key if asked by law enforcement. Otherwise the user will face a criminal charge. The Electronic Frontier Foundation (EFF[3]) is involved in a case in the Supreme Court of the United States, which may determine whether requiring suspected criminals to provide their encryption keys to law enforcement is unconstitutional. The EFF is arguing that this is a violation of the right of not being forced to incriminate oneself, as given in the Fifth Amendment.

1.2 Technical Terms

In this section basic terms used in the following chapters are explained.

Plaintext: Plaintext is information a sender wishes to transmit to a receiver. Cleartext is often used as a synonym. Before the computer era, plaintext most commonly meant message text in the language of the communicating parties. Plaintext has reference to the operation of cryptographic algorithms, usually encryption algorithms, and is the input upon which they operate. Cleartext, by contrast, refers to data that is transmitted or stored unencrypted.

Ciphertext: Ciphertext is the result of encryption performed on plaintext using an algorithm. Ciphertext is also known as encrypted or encoded information because it contains a form of the original plaintext that is unreadable by a human or computer without the proper algorithm to decrypt it.

Key: Key is a piece of information that determines the functional output of a cryptographic algorithm. Without a key, the algorithm would produce no useful result. In encryption, a key specifies the particular transformation of plaintext into ciphertext, or vice versa during

[3] The Electronic Frontier Foundation (EFF) is an international non-profit digital rights advocacy and legal organization based in the United States (www.eff.org)

decryption. Keys are also used in other cryptographic algorithms, such as digital signature schemes and message authentication codes.

Encryption – Decryption: Encryption is the process of transforming information (referred to as plaintext) using an algorithm to make it unreadable to anyone except those possessing special knowledge, usually referred to as a key. The result of the process is information (in cryptography, referred to as ciphertext). The reverse process, i.e., to make the encrypted information readable again, is referred to as decryption.

Data Integrity: Data integrity is a term used to refer to the accuracy and reliability of data. Data must be complete, with no variations or compromises from the original, to be considered reliable and accurate.

Authentication: Authentication is the act of confirming the truth of an attribute of a datum or entity. This might involve confirming the identity of a person or software program, tracing the origins of an artifact, or ensuring that a product is what its packaging and labeling claims to be.

1.3 Symmetric-key Cryptography

1.3.1 Data Encryption Standard (DES)

1.3.1.1 Overview

In 1972, the National Institute of Standards and Technology (called the National Bureau of Standards at the time) decided that a strong cryptographic algorithm was needed to protect non-classified information. The algorithm was required to be cheap, widely available, and very secure. NIST envisioned something that would be available to the general public and could be used in a wide variety of applications. So they asked for public proposals for such an algorithm. In 1974 IBM submitted the Lucifer algorithm, which appeared to meet most of NIST's design requirements.

NIST enlisted the help of the National Security Agency to evaluate the security of Lucifer. At the time many people distrusted the NSA due to their extremely secretive activities, so there was initially a certain degree of skepticism regarding the analysis of Lucifer. One of the greatest worries was that the key length, originally 128 bits, was reduced to just 56 bits, weakening it significantly.

The modified Lucifer algorithm was adopted by NIST as a federal standard on November 23, 1976. Its name was changed to the Data Encryption Standard (DES). The algorithm specification was published in January 1977, and with the official backing of the government it became a very widely employed algorithm in a short amount of time.

Unfortunately, over time various shortcut attacks were found that could significantly reduce the amount of time needed to find a DES key by brute force. And as computers became progressively faster and more powerful, it was recognized that a 56-bit key was simply not large enough for high security applications. As a result of these serious flaws, NIST abandoned their official endorsement of DES in 1997.

1.3.1.2 Algorithm Description

DES encrypts and decrypts data in 64-bit blocks, using a 64-bit key (although the effective key strength is only 56 bits). It takes a 64-bit block of plaintext as input and outputs a 64-bit block of ciphertext. Since it always operates on blocks of equal size and it uses both permutations and substitutions in the algorithm, DES is both a block cipher and a product cipher.

DES has 16 rounds, meaning the main algorithm is repeated 16 times to produce the ciphertext. It has been found that the number of rounds is exponentially proportional to the amount of time required to find a key using a brute-force attack. So as the number of rounds increases, the security of the algorithm increases exponentially.

1.3.1.3 Key Scheduling

Although the input key for DES is 64 bits long, the actual key used by DES is only 56 bits in length. The least significant (right-most) bit in each byte is a parity bit, and should be set so that there are always an odd number of 1s in every byte. These parity bits are ignored, so only the seven most significant bits of each byte are used, resulting in a key length of 56 bits.

The first step is to pass the 64-bit key through a permutation (see Table 31: *Permuted Choice 1*) to produce the 56-bit key.

Now that the 56-bit key is ready, the next step is to use this key to generate 16 48-bit subkeys, called K[1]-K[16], which are used in the 16 rounds of DES for encryption and decryption. The procedure for generating subkeys - known as key scheduling - is the following:

1. Set the round number R to 1.
2. Split the current 56-bit key, K, up into two 28-bit blocks, L (the left-hand half) and R (the right-hand half).
3. Rotate L left by the number of bits specified in the Table 33: *Subkey Rotation*, and rotate R left by the same number of bits as well.
4. Join L and R together to get the new K.
5. Apply a permutation (see Table 32: *Permuted Choice 2*) to K to get the final K[R], where R is the number of the current round.
6. Increment R by 1 and repeat the procedure until all 16 subkeys K[1]-K[16] are ready.

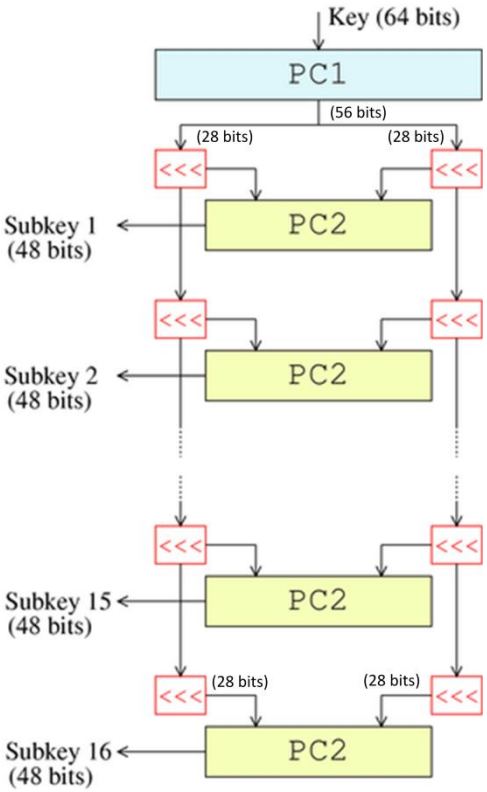


Figure 1: *DES Key Schedule*

1.3.1.4 Plaintext Preparation

Once the key scheduling has been performed, the next step is to prepare the plaintext for the actual encryption. This is done by passing the plaintext through a permutation called the Initial Permutation (see Table 34: *Initial Permutation*). This table also has an inverse, called the Inverse Initial Permutation (see Table 35: *Inverse Initial Permutation*). If you run a block of plaintext through the initial permutation and then pass the resulting block through the inverse initial permutation, you'll end up with the original block.

1.3.1.5 DES Core Function

DES core function is also known as the “Feistel function” (see Figure 2). Once the key scheduling and plaintext preparation have been completed, the actual encryption or decryption is performed by the main DES algorithm. The 64-bit block of input data is first split into two halves, L and R. L is the left-most 32 bits, and R is the right-most 32 bits. The

following process is repeated 16 times, making up the 16 rounds of standard DES. The 16 sets of halves are called L[0]-L[15] and R[0]-R[15]. Each DES round has the following steps:

1. R[i-1] - where i is the round number, starting at 1 - is taken and fed into the E-Bit Selection Table (see Table 36: *E-Bit Selection*), which is like a permutation, except that some of the bits are used more than once. This expands the number R[i-1] from 32 to 48 bits to prepare for the next step.
2. The 48-bit R[i-1] is XORed with K[i] and stored in a temporary buffer so that R[i-1] is not modified.
3. The result from the previous step is now split into 8 segments of 6 bits each. The left-most 6 bits are B[1], and the right-most 6 bits are B[8]. These blocks form the index into the S-boxes (see Table 38 to Table 45), which are used in the next step. The Substitution boxes, known as S-boxes, are a set of 8 two-dimensional arrays, each with 4 rows and 16 columns. The numbers in the boxes are always 4 bits in length, so their values range from 0-15. The S-boxes are numbered S[1] to S[8].
4. Starting with B[1], the first and last bits of the 6-bit block are taken and used as an index into the row number of S[1], which can range from 0 to 3, and the middle four bits are used as an index into the column number, which can range from 0 to 15. The number from this position in the S-box is retrieved and stored away. This is repeated with B[2] and S[2], B[3] and S[3], and the others up to B[8] and S[8]. At this point, you now have 8 4-bit numbers, which when strung together one after the other in the order of retrieval, give a 32-bit result.
5. The result from the previous stage is now passed into the P Permutation (see Table 37: *P Permutation*).
6. This number is now XORed with L[i-1], and moved into R[i]. R[i-1] is moved into L[i].
7. At this point the new L[i] and R[i] are ready. Here, i is incremented and the core function is repeated until i = 17, which means that 16 rounds have been executed and keys K[1]-K[16] have all been used.

When L[16] and R[16] have been obtained, they are joined back together in the same fashion they were split apart (L[16] is the left-hand half, R[16] is the right-hand half), then the two halves are swapped, R[16] becomes the left-most 32 bits and L[16] becomes the right-most 32 bits of the pre-output block and the resultant 64-bit number is called the pre-output.

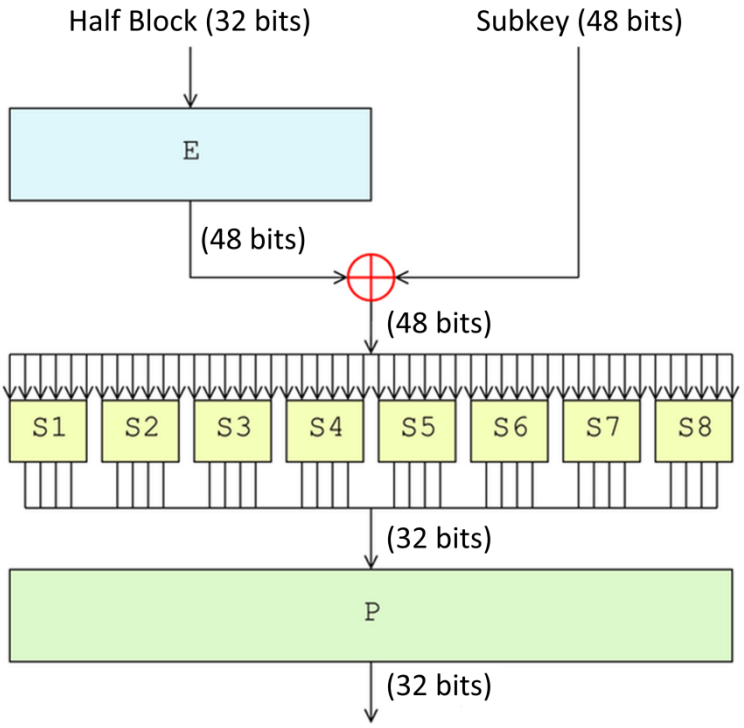


Figure 2: *Feistel Function*

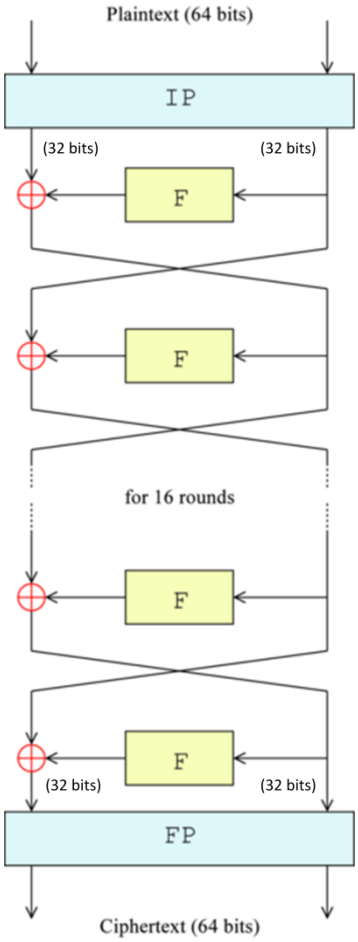


Figure 3: *Main Process of DES*

1.3.1.6 Ciphertext Preparation

The final step is to apply the inverse initial permutation at the pre-output. The result is the completely encrypted ciphertext.

1.3.1.7 Decryption

The same algorithm can be used for encryption or decryption. The method described above will encrypt a block of plaintext and return a block of ciphertext. In order to decrypt the ciphertext and get the original plaintext again, the procedure is simply repeated but the subkeys are applied in reverse order, i.e. K[16] to K[1]. Other than that, decryption is performed exactly in the same way as encryption.

1.3.2 Advanced Encryption Standard (AES)

1.3.2.1 Overview

The Advanced Encryption Standard (AES) is a specification for the encryption of electronic data established by the U.S. National Institute of Standards and Technology (NIST) in 2002. Originally called Rijndael, the algorithm was developed by two Belgian cryptographers, Joan Daemen and Vincent Rijmen, who submitted to the AES selection process. Strictly speaking, the AES standard is a variant of Rijndael where the block size is restricted to 128 bits.

AES has been adopted by the U.S. government and is now used worldwide. It supersedes the Data Encryption Standard (DES). The algorithm described by AES is a symmetric-key algorithm, meaning the same key is used for both encrypting and decrypting the data.

In the United States, AES was announced by the NIST as U.S. FIPS PUB 197 (FIPS[4] 197) on November 26, 2001. This announcement followed a five-year standardization process in which fifteen competing designs were presented and evaluated, before the Rijndael algorithm was selected as the most suitable. It became effective as a federal government standard on May 26, 2002 after approval by the Secretary of Commerce. AES is included in the ISO/IEC 18033-3 standard. AES is available in many different encryption packages, and is the first publicly accessible and open algorithm approved by the National Security Agency (NSA) for top secret information when used in an NSA approved cryptographic module.

[4] A Federal Information Processing Standard (FIPS) is a publicly announced standardization developed by the United States federal government for use in computer systems by all non-military government agencies and by government contractors, when properly invoked and tailored on a contract.

The main process of AES is depicted in Figure 4.

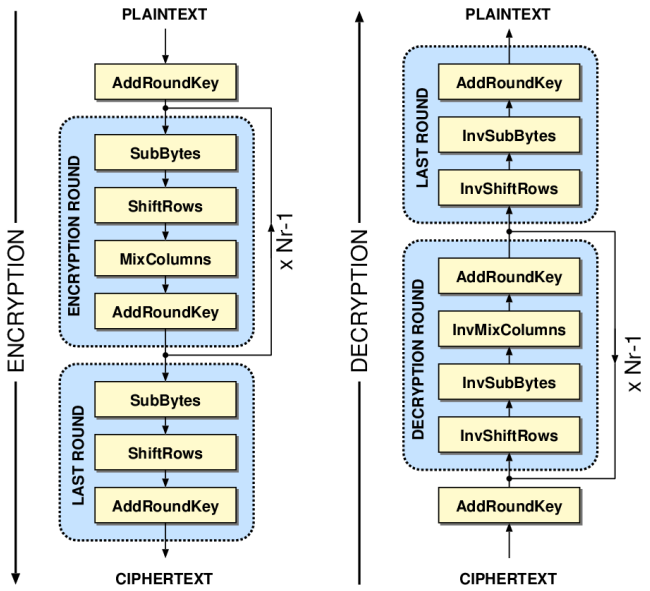


Figure 4: *AES Main Process*

1.3.2.2 Rijndael Key Schedule

AES (Rijndael) uses a key schedule to expand a short key into a number of separate round keys. This is known as the Rijndael key schedule. Rijndael's key schedule utilizes a number of operations, which will be described before describing the key schedule.

- Rotate: The rotate operation takes a 32-bit word and rotates it eight bits to the left such that the high eight bits "wrap around" and become the low eight bits of the result.
- Rcon: Rcon is what the Rijndael documentation calls the exponentiation of 2 to a user-specified value. Note that, this operation is not performed with regular integers, but in Rijndael's finite field. The Rcon can be computed using a specific vector (see Table 46: *Rcon[256]*).
- S-box: The Rijndael S-box is a matrix (square array of numbers) used in the Rijndael cipher. The S-box (substitution box) serves as a lookup table (see Table 47: *Rijndael S-Box*).

1.3.2.3 Key Schedule Core

This operation is used as an inner loop in the key schedule, and is done thus:

- The input is a 32-bit word and at an iteration number i. The output is a 32-bit word.
- Copy the input over to the output.
- Use the above described rotate operation to rotate the output eight bits to the left.
- Apply Rijndael's S-box on all four individual bytes in the output word.
- On just the first (leftmost) byte of the output word, exclusive or the byte with 2 to the power of (i-1). In other words, perform the Rcon operation with i as the input, and exclusive or the Rcon output with the first byte of the output word.

Since the key schedule for 128-bit, 192-bit, and 256-bit encryption are very similar, with only some constants changed, the following key size constants are defined here:

- n has a value of 16 for 128-bit keys, 24 for 192-bit keys, and 32 for 256-bit keys
- b has a value of 176 for 128-bit keys, 208 for 192-bit keys, and 240 for 256-bit keys

Rijndael's key schedule is done as follows:

1. The first n bytes of the expanded key are the encryption key.
2. The Rcon iteration value i, is set to 1.
3. Until b bytes of expanded key are produced, the following procedure is executed to generate n more bytes of expanded key:
    - The following steps are performed to create 4 bytes of expanded key:
        i. Create a 4-byte temporary variable, t.
        ii. Assign the value of the previous four bytes in the expanded key to t.
        iii. Perform the key schedule core (see Key Schedule Core) on t, with i as the Rcon iteration value.
        iv. Increment i by 1.
        v. Exclusive-or t with the four-byte block n bytes before the new expanded key. This becomes the next 4 bytes in the expanded key.
    - Then, the following steps are performed three times to create the next twelve bytes of expanded key:
        i. Assign the value of the previous 4 bytes in the expanded key to t.
        ii. Exclusive-or t with the four-byte block n bytes before the new expanded key. This becomes the next 4 bytes in the expanded key.

- If a 256-bit key is generated, the following steps are performed to generate the next 4 bytes of expanded key:
    i. Assign the value of the previous 4 bytes in the expanded key to t.
    ii. Run each of the 4 bytes in t through Rijndael's S-box.
    iii. Exclusive-or t with the 4-byte block n bytes before the new expanded key. This becomes the next 4 bytes in the expanded key.
- If a 128-bit key is generated, the following steps are not performed. If a 192-bit key is generated, the following steps are performed twice. If a 256-bit key is generated, the following steps are performed three times:
    i. Assign the value of the previous 4 bytes in the expanded key to t.
    ii. Exclusive-or t with the four-byte block n bytes before the new expanded key. This becomes the next 4 bytes in the expanded key.

1.3.2.4 Description of the Algorithm

AES is based on a design principle known as a substitution-permutation network, and is fast in both software and hardware. Unlike its predecessor DES, AES does not use a Feistel network. AES is a variant of Rijndael which has a fixed block size of 128 bits, and a key size of 128, 192, or 256 bits. By contrast, the Rijndael specification per se is specified with block and key sizes that may be any multiple of 32 bits, both with a minimum of 128 and a maximum of 256 bits.

AES operates on a 4×4 column-major order matrix of bytes, termed the state, although some versions of Rijndael have a larger block size and have additional columns in the state. Most AES calculations are done in a special finite field.

The key size used for an AES algorithm specifies the number of repetitions of transformation rounds that convert the input, called the plaintext, into the final output, called the ciphertext. The number of cycles of repetition is as follows:

- 10 cycles of repetition for 128 bit keys
- 12 cycles of repetition for 192 bit keys
- 14 cycles of repetition for 256 bit keys

Each round consists of several processing steps, including one that depends on the encryption key itself.

1. Key Expansion: round keys are derived from the cipher key using Rijndael's key schedule
2. Initial Round:
    i. AddRoundKey: each byte of the state is combined with the round key using bitwise xor
3. Rounds:
    i. SubBytes: a non-linear substitution step where each byte is replaced with another according to a lookup table
    ii. ShiftRows: a transposition step where each row of the state is shifted cyclically a certain number of steps
    iii. MixColumns: a mixing operation which operates on the columns of the state, combining the four bytes in each column
    iv. AddRoundKey
4. Final Round (no MixColumns)
    i. SubBytes
    ii. ShiftRows
    iii. AddRoundKey

1.3.2.5 The SubBytes Step

In the SubBytes step, each byte in the state matrix is replaced with a SubByte using an 8-bit substitution box, the Rijndael S-box (see Table 47: *Rijndael S-Box*). This operation provides the non-linearity in the algorithm. The S-box used is derived from the multiplicative inverse over GF(2^8), known to have good non-linearity properties. 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 so is a derangement), and also any opposite fixed points.

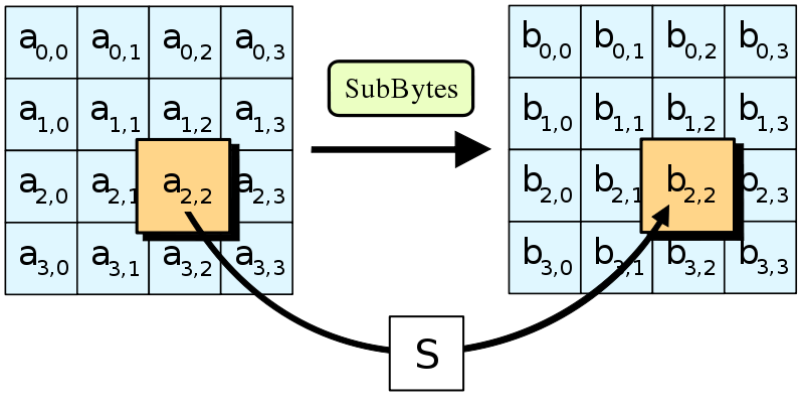


Figure 5: *AES SubBytes*

1.3.2.6 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. For AES, 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. For blocks of sizes 128 bits and 192 bits, the shifting pattern is the same. Row n is shifted left circular by n-1 bytes. In this way, each column of the output state of the ShiftRows step is composed of bytes from each column of the input state. (Rijndael variants with a larger block size have slightly different offsets). For a 256-bit block, the first row is unchanged and the shifting for the second, third and fourth row is 1 byte, 3 bytes and 4 bytes respectively—this change only applies for the Rijndael algorithm when used with a 256-bit block, as AES does not use 256-bit blocks.

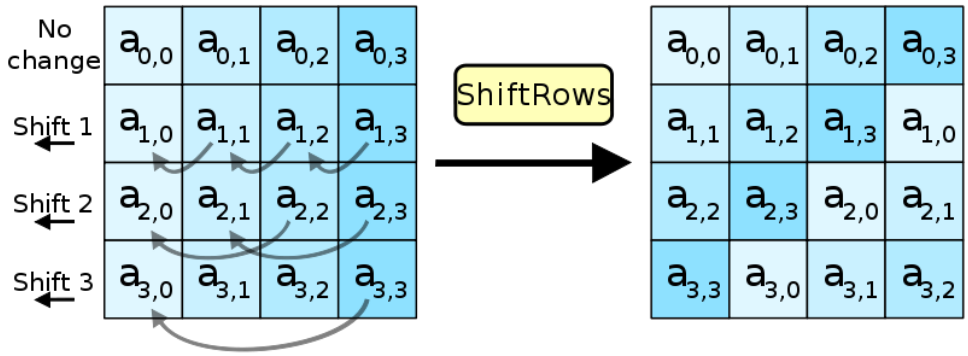


Figure 6: *AES ShiftRows*

1.3.2.7 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 provides diffusion in the algorithm.

During this operation, each column is multiplied by a known matrix (see Table 1).

Table 1: *MixColumns Multiplication Matrix*

2	3	1	1
1	2	3	1
1	1	2	3
3	1	1	2

The multiplication operation is defined as multiplication by 1 means no change, multiplication by 2 means shifting to the left, and multiplication by 3 means shifting to the left and then performing xor with the initial unshifted value. After shifting, a conditional xor with 0x11B should be performed if the shifted value is larger than 0xFF.

In more general sense, each column is treated as a polynomial over $GF(2^8)$ and is then multiplied modulo x^4+1 with a fixed polynomial $c(x) = 0x03 \cdot x^3 + x^2 + x + 0x02$. The coefficients are displayed in their hexadecimal equivalent of the binary representation of bit polynomials from GF(2)[x]. The MixColumns step can also be viewed as a multiplication by a particular MDS matrix in a finite field.

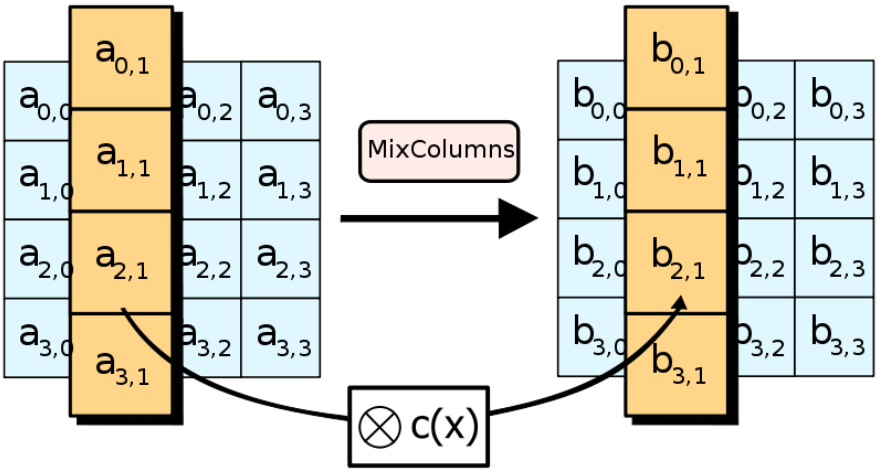


Figure 7: *AES MixColumns*

1.3.2.8 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 Rijndael's key schedule; each subkey is 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.

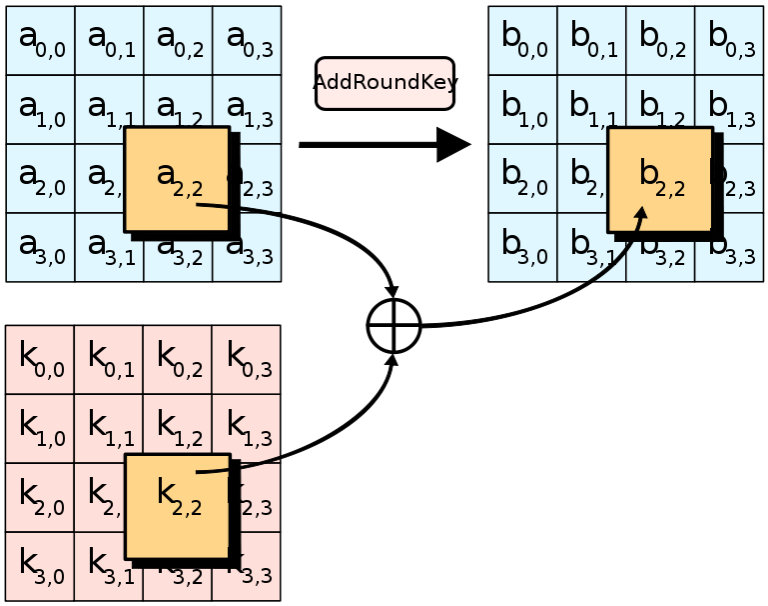


Figure 8: *AES AddRoundKey*

1.3.2.9 Decryption

The above process describes the way encryption is performed. Decryption is almost the same process but the steps (SubBytes, ShiftRows, MixColumns, and AddRoundKey) are executed in a different order using different tables. Details of the way that decryption process is implemented can be found in section 3.3.4).

1.3.3 International Data Encryption Algorithm (IDEA)

1.3.3.1 Overview

The block cipher IDEA was first presented by Xuejia Lai and James Massey of the Swiss Federal Institute of Technology in 1990 and was then called PES (Proposed Encryption Standard). In 1991 after Biham and Shamir presented their results regarding differential cryptanalysis, the authors developed an improved version of the PES algorithm to increase the security against this attack and the new algorithm was called IPES (Improved Proposed Encryption Standard) while finally in 1992 its name was changed officially to IDEA.

The IDEA is a symmetric, block oriented encryption algorithm, which operates on a 64-bit plaintext and uses a 128 bit length key. The substitution boxes and the associated "lookup tables" used in the rest block ciphers available to-date (and among them DES) have been completely dispensed with. The required confusion in this algorithm is achieved by successively using three different and incompatible group operations on pairs of 16-bit sub blocks and mixing them (in such a way that at no point in the encryption process the same algebraic operation is used contiguously) while the structure of the cipher was carefully chosen to provide the necessary diffusion requirement. These three algebraic operations are the following:

- Bitwise XOR (denoted with $\oplus$)
- Addition of integers modulo (2^{16}) with inputs and outputs treated as unsigned 16-bit integers (denoted with $\boxplus$)
- Multiplication of integers modulo ($2^{16}+1$) with inputs and outputs treated as unsigned 16-bit integers (This operation can be also viewed as IDEA's equivalent S-box) (denoted with $\odot$)

All these operations operate on 16-bit sub-blocks. Their use in combination provides for a complex transformation of the input making cryptanalysis much more difficult than with an algorithm such as e.g. DES, which relies solely on the XOR function.

IDEA uses a 128 bit key which is double the key size of DES, making it highly immune to attacks. IDEA uses algebraic operations completely and it entirely avoids the use of any lookup tables or S-boxes. The strength of IDEA lies in its modulo multiplication operations. The working of IDEA can be visualized as—the 64-bit plain text block is divided into 4 portions of plain text (each of size 16 bits), say P1 to P4. Thus, P1 to P4 are the inputs for the first round of the algorithm. There are 8 such rounds. In each round, 6 subkeys (each of size 16 bits) are generated from the original 128 bit key. These subkeys are applied to the 4 input blocks P1 to P4. Thus, for the 1st round there are 6 subkeys K1 to K6. For the 2nd round, there are keys K7 to K12. Finally, keys K43 to K48 will be used. The final step consists of an Output Transformation, which uses just 4 subkeys. The final output produced is the output produced by the Output Transformation round.

The main process of IDEA is depicted in Figure 9.

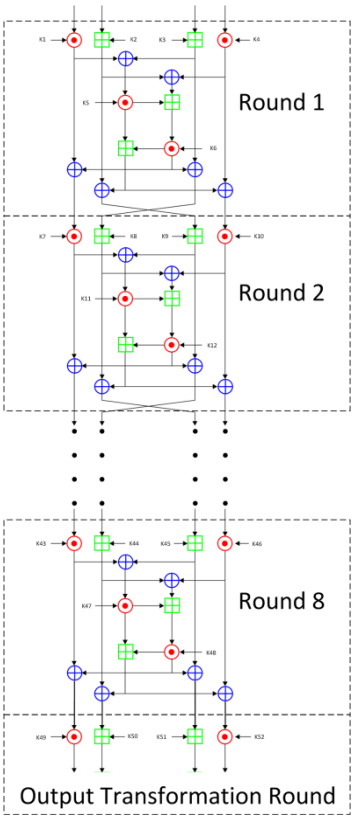


Figure 9: *IDEA Main Process*

The designers analyzed IDEA to measure its strength against differential cryptanalysis and concluded that it is immune under certain assumptions. No successful linear or algebraic weaknesses have been reported. As of 2007, the best attack which applies to all keys can break IDEA reduced to 6 rounds (the full IDEA cipher uses 8.5 rounds). Note that a "break" is any attack which requires less than 2128 operations; the 6-round attack requires 264 known plaintexts and 2126.8 operations.

The very simple key schedule makes IDEA subject to a class of weak keys; some keys containing a large number of 0 bits produce weak encryption. These are of little concern in practice, being sufficiently rare that they are unnecessary to avoid explicitly when generating keys randomly. A simple fix was proposed; exclusive-ORing each subkey with a 16-bit constant, such as 0x0DAE. Larger classes of weak keys were found in 2002.

1.3.3.2 Key Generation

The initial 6 subkeys K1 to K6 are generated from the original 128 bit key. Since the sub -keys consist of 16 bits each, out of the original 128 bits, the first 96 bits are used for the first round. Thus, at the end of the first round, bits 97–128 of the original key are unused. In the second round, the unused 32 bits of the first round are used. To generate the rest of the sub -keys for the second round, 64 more bits are required. This is obtained by shifting the original key left circularly by 25 bits. Then, the modified key is now used to generate the rest of the 4 subkeys in the same way as the first round keys are generated. The same is done for the subkey generation for the rest of the rounds.

1.3.3.3 Encryption Round

In each round of the 8 rounds of algorithm, the following sequence of events is performed:

1. Multiply P1 and K1
2. Add P2 and K2
3. Add P3 and K3
4. Multiply P4 and K4
5. XOR the results of step 1 and step 3
6. XOR the results of step 2 and step 4

7. Multiply the results of step 5 with K5
8. Add the results of step 6 and step 7
9. Multiply the results of step 8 with K6
10. Add the results of step 7 and step 9
11. XOR the results of step 1 and step 9
12. XOR the results of step 3 and step 9
13. XOR the results of step 2 and step 10
14. XOR the results of step 4 and step 10

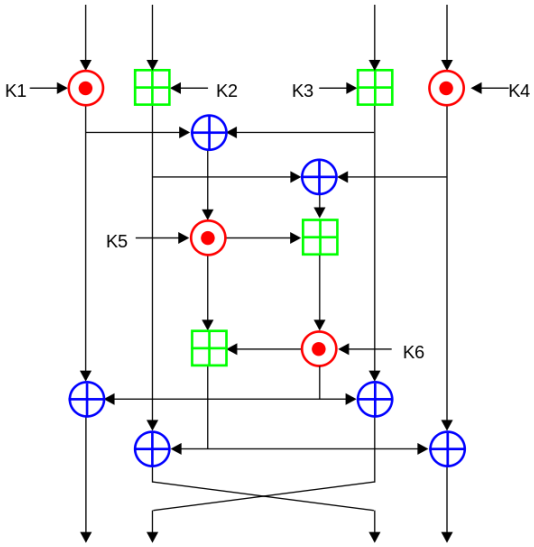


Figure 10: *IDEA Round*

Sequence of events followed in the output transformation round:

1. Multiply R1 and K1
2. Add R2 and K2
3. Add R3 and K3
4. Multiply R4 and K4

The outputs of the round are given in the same order to the next round. After the 8th round, the inner 2 blocks are swapped and given as input to the final transformation round. Finally, the four sub -blocks are attached to get the final encrypted result.

1.3.3.4 Decryption

Decryption uses exactly the same sequence of operations of successive 64-bit blocks of the ciphertext, but with a different set of subkeys. The same 52 key sub-blocks generated for encryption are rearranged and inverted accordingly to produce the decryption key schedule. Those that are added are replaced by their two's complement. Those that are multiplied in are replaced by their multiplicative inverse, modulo $2^{16}+1$, but those used to calculate the cross-footed F-functions are not changed. Keys XORed in would not need to be changed, but there aren't any such keys in IDEA.

The decryption sub -keys (relative to the encryption subkeys s1 to s52) are generated as shown in Table 2.

Table 2: *Decryption Subkeys Generation Table*

1st round	s49*	s50#	s51#	s52*	s47	s48
2nd round	s43*	s45#	s44#	s46*	s41	s42
3rd round	s37*	s39#	s38#	s39*	s35	s36
4th round	s31*	s33#	s32#	s34*	s29	s30
5th round	s25*	s27#	s26#	s28*	s23	s24
6th round	s19*	s21#	s20#	s22*	s17	s18
7th round	s13*	s15#	s14#	s16*	s11	s12
8th round	s7*	s9#	s8#	s10*	s5	s6
Final transformation	-	-	s1*	s2#	s3#	s4*

- sXX* = multiplicative inverse of sXX modulus $2^{16}+1$
- sXX# = additive inverse of sXX modulus 2^{16}

1.3.4 Block Cipher Operation Modes

1.3.4.1 Introduction

In cryptography, a mode of operation is the procedure of enabling the repeated and secure use of a block cipher under a single key. A block cipher by itself allows encryption only of a single data block of the cipher's block length. When targeting a variable-length message, the data must first be partitioned into separate cipher blocks. Typically, the last block must also

be extended to match the cipher's block length using a suitable padding scheme. A mode of operation describes the process of encrypting each of these blocks, and generally uses randomization based on an additional input value, often called an initialization vector, to allow doing so safely.

Modes of operation have primarily been defined for encryption and authentication. Historically, encryption modes have been studied extensively in regard to their error propagation properties under various scenarios of data modification. Later development regarded integrity protection as an entirely separate cryptographic goal from encryption. Some modern modes of operation combine encryption and authentication in an efficient way, and are known as authenticated encryption modes.

An initialization vector (IV) is a block of bits that is used by several modes to randomize the encryption and hence to produce distinct ciphertexts even if the same plaintext is encrypted multiple times, without the need for a slower re-keying process.

An initialization vector has different security requirements than a key, so the IV usually does not need to be secret. However, in most cases, it is important that an initialization vector is never reused under the same key. For CBC and CFB, reusing an IV leaks some information about the first block of plaintext, and about any common prefix shared by the two messages. For OFB and CTR, reusing an IV completely destroys security. In CBC mode, the IV must, in addition, be unpredictable at encryption time; in particular, the (previously) common practice of re-using the last ciphertext block of a message as the IV for the next message is insecure (for example, this method was used by SSL 2.0). If an attacker knows the IV (or the previous block of ciphertext) before he specifies the next plaintext, he can check his guess about plaintext of some block that was encrypted with the same key before (this is known as the TLS CBC IV attack).

As a special case, if the plaintexts are always small enough to fit into a single block (with no padding), then with some modes (ECB, CBC, PCBC), re-using an IV will leak only whether two plaintexts are equal. This can be useful in cases where one wishes to be able to test for equality without decrypting or separately storing a hash.

1.3.4.2 Electronic Codebook (ECB mode)

The simplest of the encryption modes is the electronic codebook (ECB) mode. The message is divided into blocks and each block is encrypted separately.

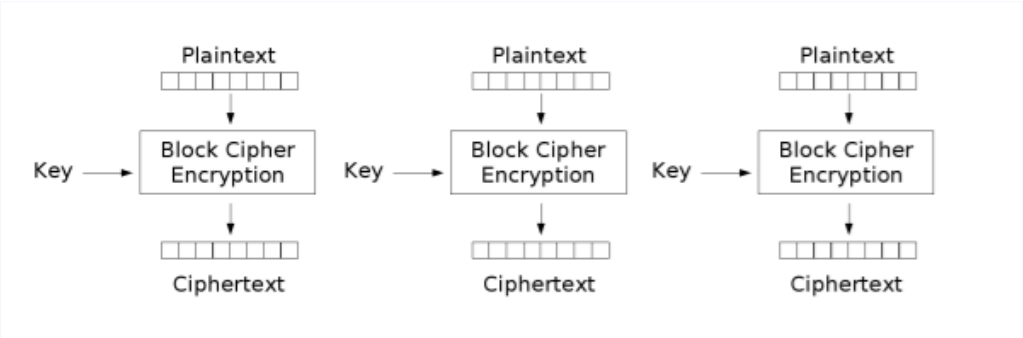


Figure 11: *Electronic Codebook (ECB) Encryption*

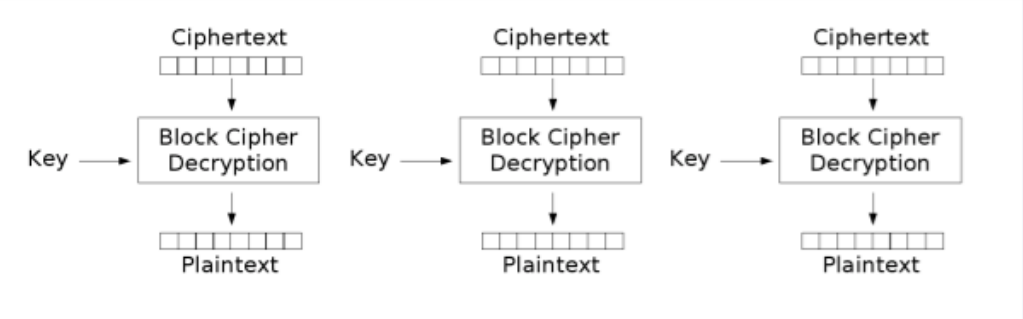
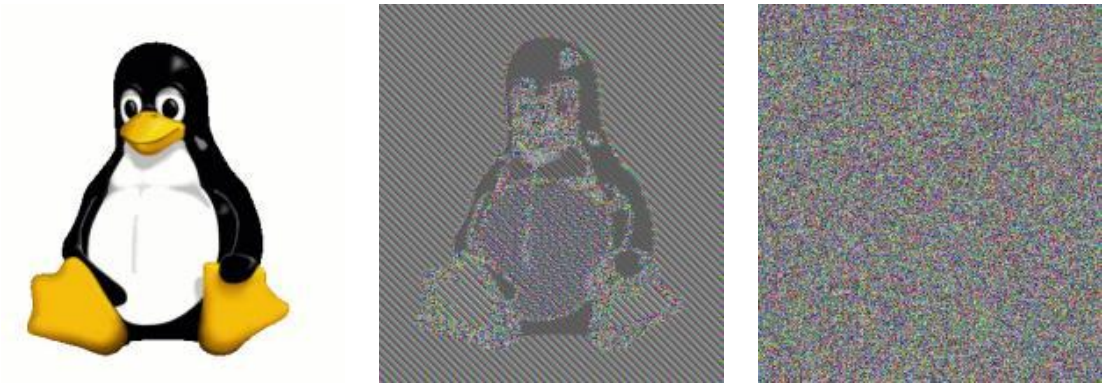


Figure 12: *Electronic Codebook (ECB) Decryption*

The disadvantage of this method is that identical plaintext blocks are encrypted into identical ciphertext blocks; thus, it does not hide data patterns well. In some senses, it doesn't provide serious message confidentiality, and it is not recommended for use in cryptographic protocols at all.

A striking example of the degree to which ECB can leave plaintext data patterns in the ciphertext can be seen when ECB mode is used to encrypt a bitmap image which uses large areas of uniform color. While the color of each individual pixel is encrypted, the overall image may still be discerned as the pattern of identically colored pixels in the original remains in the encrypted version (see Figure 13).



Original Image *Encrypted using ECB mode* *Encrypted using modes other than ECB*

Figure 13: *Difference of ECB Mode from the Others*

1.3.4.3 Cipher-block Chaining (CBC mode)

IBM invented the cipher-block chaining (CBC) mode of operation in 1976. In CBC mode, each block of plaintext is XORed with the previous ciphertext block before being encrypted. This way, each ciphertext block depends on all plaintext blocks processed up to that point. To make each message unique, an initialization vector must be used in the first block.

Encryption and decryption algorithms are as follows:

- $C_i = E_K(P_i \oplus C_{i-1})$, $C_0 = IV$
- $P_i = D_K(C_i) \oplus C_{i-1}$, $C_0 = IV$

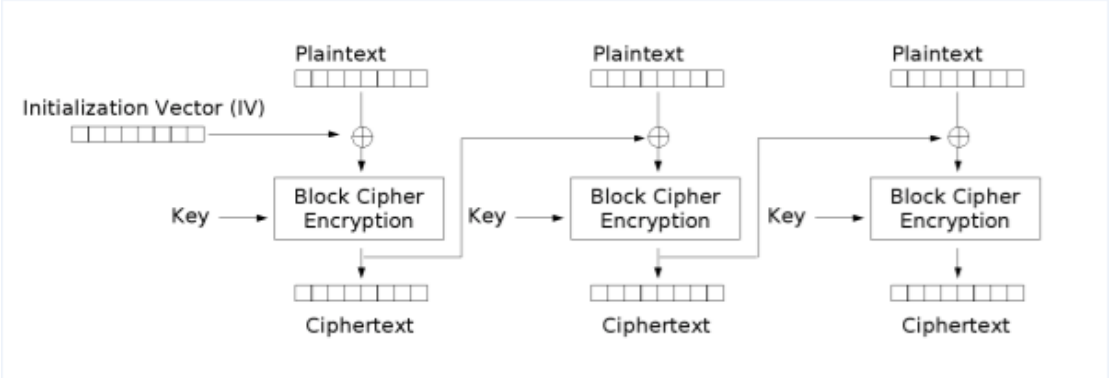


Figure 14: *Cipher-block Chaining Encryption*

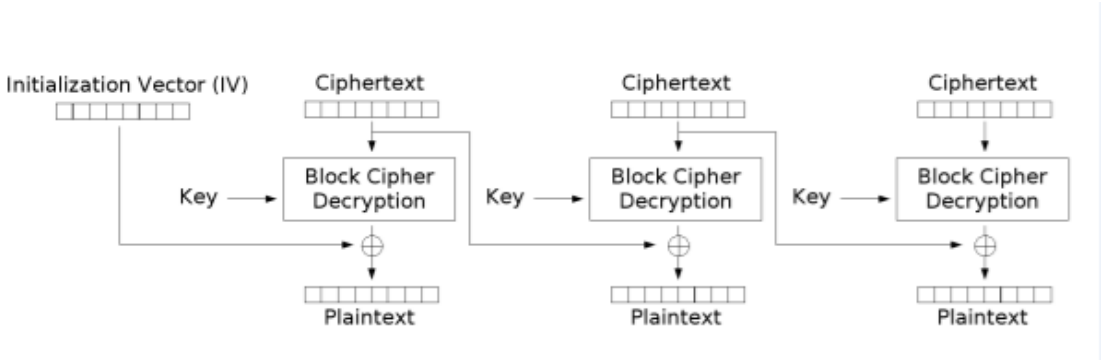


Figure 15: *Cipher-block Chaining Decryption*

CBC has been the most commonly used mode of operation. Its main drawbacks are that encryption is sequential (i.e., it cannot be parallelized), and that the message must be padded to a multiple of the cipher block size. One way to handle this last issue is through the method known as ciphertext stealing. Note that a one-bit change in a plaintext or IV affects all following ciphertext blocks.

Decrypting with the incorrect IV causes the first block of plaintext to be corrupt but subsequent plaintext blocks will be correct. This is because a plaintext block can be recovered from two adjacent blocks of ciphertext. As a consequence, decryption can be parallelized. Note that a one-bit change at the ciphertext causes complete corruption of the corresponding block of plaintext and inverts the corresponding bit in the following block of plaintext, but the rest of the blocks remain intact.

1.3.4.4 Propagating Cipher-block Chaining (PCBC mode)

The propagating cipher-block chaining or plaintext cipher-block chaining mode was designed to cause small changes in the ciphertext to propagate indefinitely when decrypting, as well as when encrypting.

Encryption and decryption algorithms are as follows:

- $C_i = E_K(P_i \oplus P_{i-1} \oplus C_{i-1})$, $P_0 \oplus C_0 = IV$
- $P_i = D_K(C_i) \oplus P_{i-1} \oplus C_{i-1}$, $P_0 \oplus C_0 = IV$

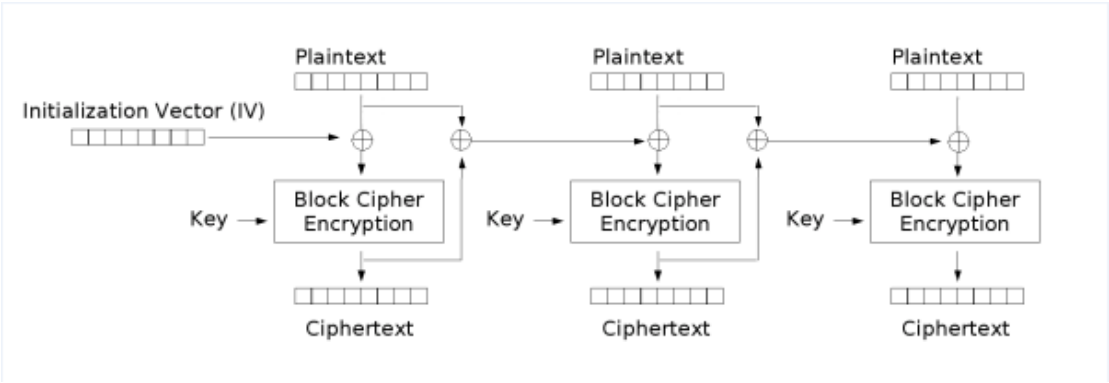


Figure 16: *Propagating Cipher-block Chaining (PCBC) Encryption*

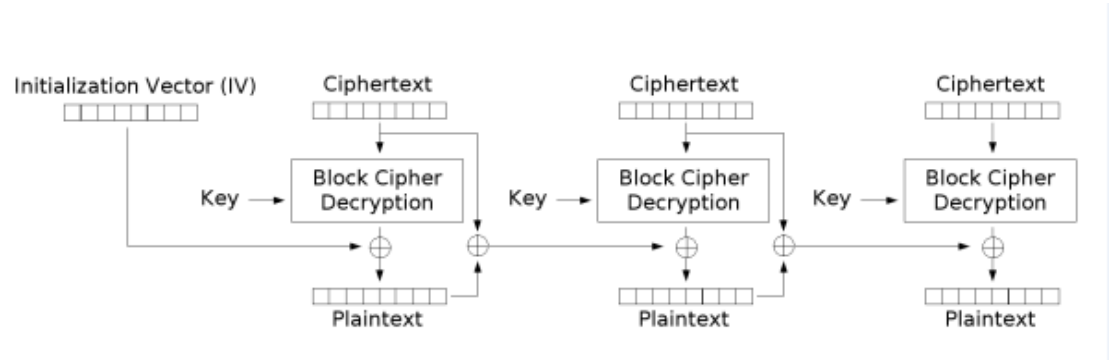


Figure 17: *Propagating Cipher-block Chaining (PCBC) Decryption*

1.3.4.5 Cipher Feedback (CFB mode)

The cipher feedback (CFB) mode, a close relative of CBC, makes a block cipher into a self-synchronizing stream cipher. Operation is very similar; in particular, CFB decryption is almost identical to CBC encryption performed in reverse:

- $C_i = E_K(C_{i-1}) \oplus P_i$
- $P_i = E_K(C_{i-1}) \oplus C_i$
- $C_0 = IV$

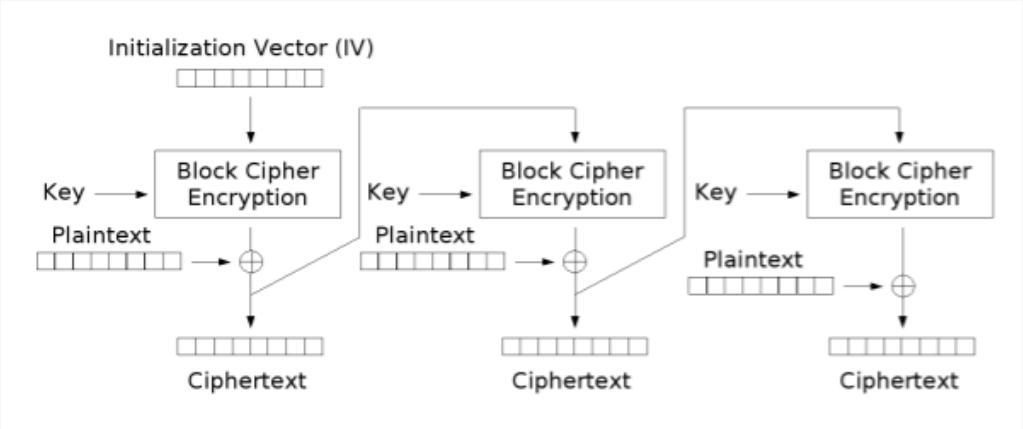


Figure 18: *Cipher Feedback (CFB) Encryption*

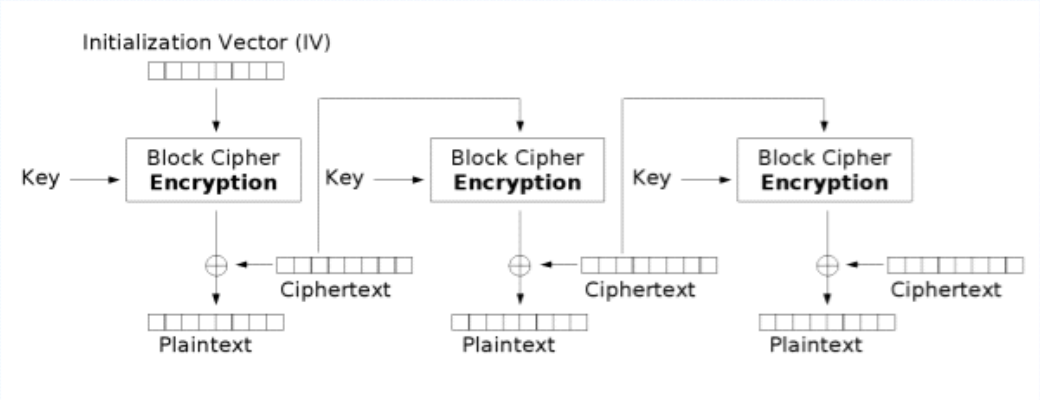


Figure 19: *Cipher Feedback (CFB) Decryption*

This simplest way of using CFB described above is not any more self-synchronizing than other cipher modes like CBC. If a whole block size of ciphertext is lost both CBC and CFB will synchronize, but losing only a single byte or bit will permanently throw off decryption. To be able to synchronize after the loss of only a single byte or bit, a single byte or bit must be encrypted at a time. CFB can be used this way when combined with a shift register as the input for the block cipher.

Like CBC mode, changes in the plaintext propagate forever in the ciphertext, and encryption cannot be parallelized. Also like CBC, decryption can be parallelized. When decrypting, a one-bit change in the ciphertext affects two plaintext blocks; a one-bit change in the corresponding plaintext block, and complete corruption of the following plaintext block. Later plaintext blocks are decrypted normally.

CFB shares two advantages over CBC mode with the stream cipher modes OFB and CTR; the block cipher is only ever used in the encrypting direction, and the message does not need to be padded to a multiple of the cipher block size (though ciphertext stealing can also be used to make padding unnecessary).

1.3.4.6 Output Feedback (OFB mode)

The output feedback (OFB) mode makes a block cipher into a synchronous stream cipher. It generates key stream blocks, which are then XORed with the plaintext blocks to get the ciphertext. Just as with other stream ciphers, flipping a bit in the ciphertext produces a flipped bit in the plaintext at the same location. This property allows many error correcting codes to function normally even when applied before encryption.

Because of the symmetry of the XOR operation, encryption and decryption are exactly the same:

- $C_j = P_j \oplus O_j$
- $P_j = C_j \oplus O_j$
- $O_j = E_K(I_j)$
- $I_j = O_{j-1}$
- $I_0 = IV$

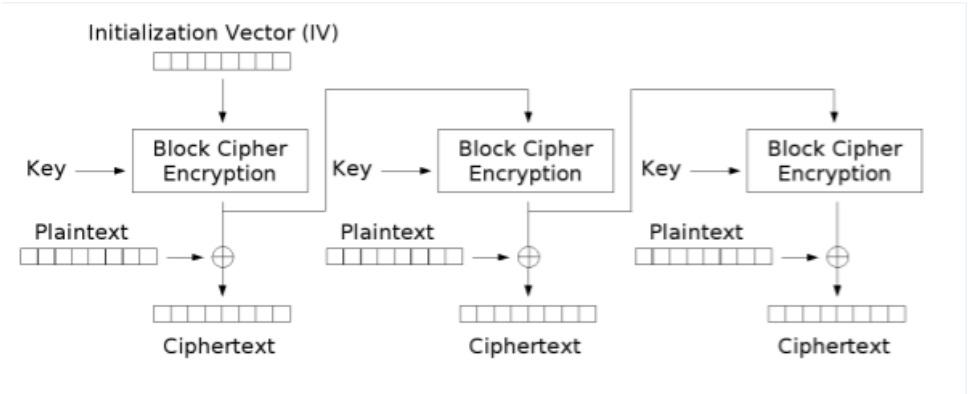


Figure 20: *Output Feedback (OFB) Encryption*

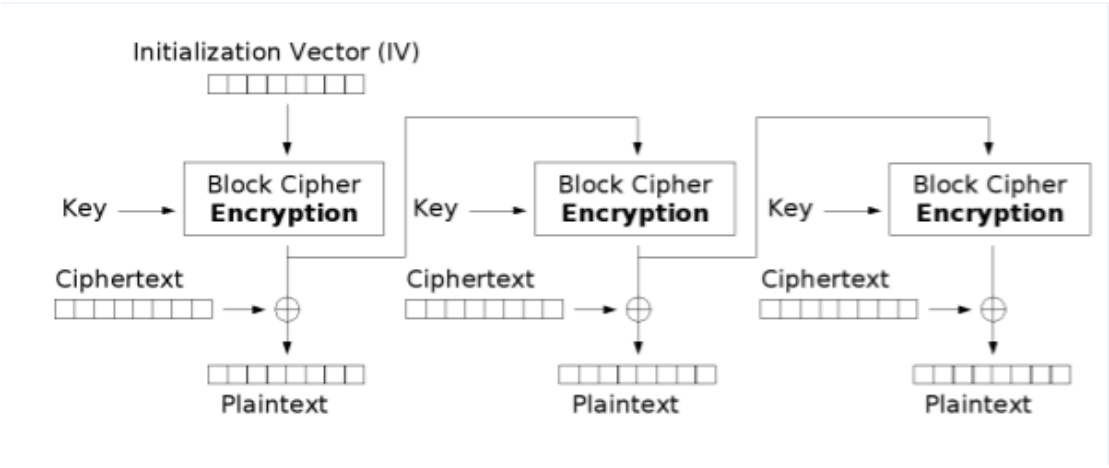


Figure 21: *Output Feedback (OFB) Decryption*

Each output feedback block cipher operation depends on all previous ones, and so cannot be performed in parallel. However, because the plaintext or ciphertext is only used for the final XOR, the block cipher operations may be performed in advance, allowing the final step to be performed in parallel once the plaintext or ciphertext is available.

It is possible to obtain an OFB mode key stream by using CBC mode with a constant string of zeroes as input. This can be useful, because it allows the usage of fast hardware implementations of CBC mode for OFB mode encryption.

1.3.4.7 Counter (CTR mode)

Like OFB, counter mode turns a block cipher into a stream cipher. It generates the next key stream block by encrypting successive values of a "counter". The counter can be any function which produces a sequence which is guaranteed not to repeat for a long time, although an actual increment-by-one counter is the simplest and most popular. By now, CTR mode is widely accepted, and problems resulting from the input function are recognized as a weakness of the underlying block cipher instead of the CTR mode. Nevertheless, there are specialized attacks like a Hardware Fault Attack that is based on the usage of a simple counter function as input.

CTR mode has similar characteristics to OFB, but also allows a random access property during decryption. CTR mode is well suited to operation on a multi-processor machine

where blocks can be encrypted in parallel. Furthermore, it does not suffer from the short-cycle problem that can affect OFB.

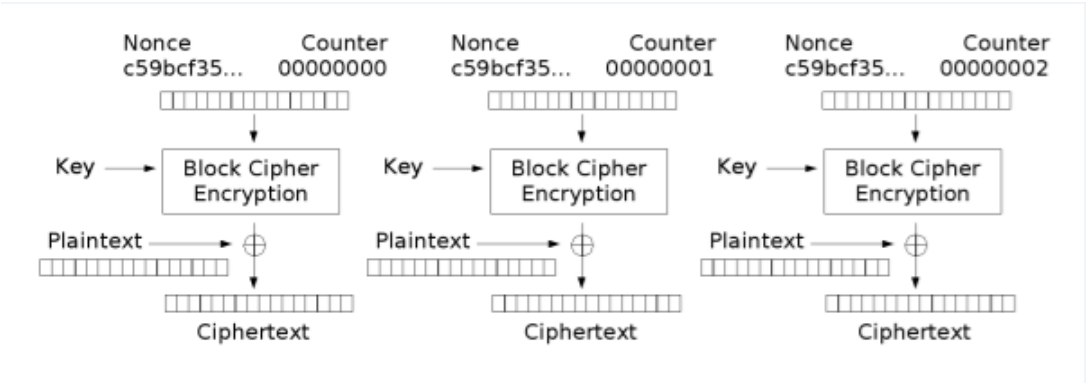


Figure 22: *Counter (CTR) Encryption*

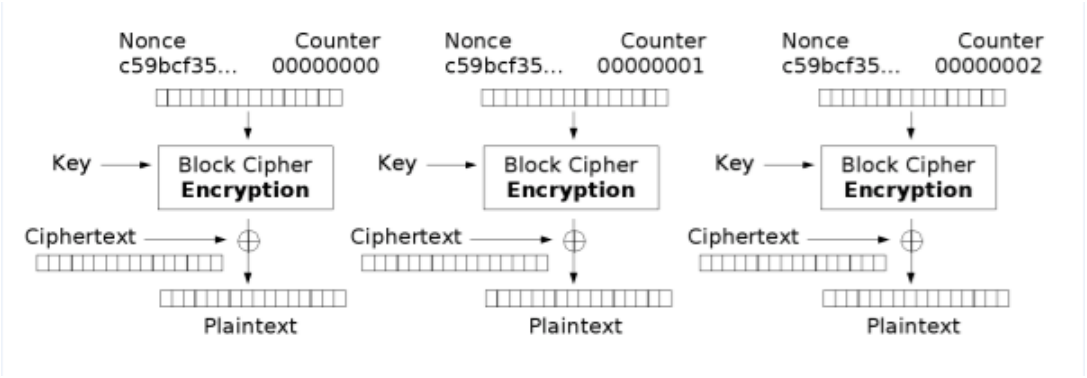


Figure 23: *Counter (CTR) Decryption*

1.3.5 Applications

1.3.5.1 DES Applications

The DES devices are used by the federal department and other government agencies for cryptographic protection of classified information. The federal government standardizes DES and specifies interoperability and security-related requirements for using encryption at the Physical Layer of the ISO Open Systems Interconnection (OSI) Reference Model in telecommunications systems conveying digital information.

Data encryption (and particularly DES) is primarily applied in:

- Electronic financial transactions: Automatic Teller Machines (devices limited to the issuance of cash or travelers checks, acceptance of deposits, or account balance reporting)
- Secure data communications, paving the road for e-commerce
- Secure video surveillance systems
- Encrypted data storage and proprietary software protection
- Access control: Software or hardware which protects passwords or Personal Identification Numbers (PINs) against unauthorized access.

DES is used in gateways to ensure privacy of user data. Also provides secure digital voice encryption in hand-held communication devices such as land mobile radio and dispatch control consoles. Data encryption through DES and is prevalent in fax machines. This allows secure data transfer over phone lines and prevents active interception of one's faxes at the receiver end, which is prevented by password entry by the user for fax retrieval. Networking applications use DES to provide network protection through data privacy, data integrity, access control and authentication. Message and file security, user authentication, secure remote system logon, and multilevel system access require data encryption, and DES algorithm is the most prevalent.

There is a need for control and access between different entities in a company's business environment, to provide secure communication between remote offices, business partners, customers, and travelling and telecommuting employees. Transmitting messages over the existing Internet backbone poses risks. VPNs were introduced to tackle exactly these issues to provide a company owned and managed network architecture. These networks provide scalable and comprehensive solutions by utilizing existing Internet backbone with additional hardware and software solutions. Strong data encryption is necessary to extend security and control features for which DES is the most commonly used. This provides secure network traffic through data privacy, data integrity, access control and authenticating entities by providing a gateway to each point of access into the business.

DES algorithm has been used for cell payload encryption in ATMs. The cryptographic units heighten security interfaces between a secure LAN and a public network. As data crosses this interface, the system encrypts each ATM cell's payload without affecting the header. Encrypted cells pass through the public network infrastructure and are decrypted upon

arriving at the destination LAN. The benefit is that the user can conduct business as usual within the LAN and can encrypt the data as it enters the non-secure public network or non-secure area of a LAN. The system provides privacy and access control guarantees when using public ATM networks.

Data security in e-Commerce applications is required to have secure website, conduct financial transactions over the Internet, authentication of users to Intranets and Extranets, secure messaging, and secure storage of digital signature keys for signature generation and verification for digital documents.

Smartcard solutions are used in wireless communication, loyalty systems, banking Pay TV and government ID. These are used to provide strong authentication in e-business. These solutions are used with standard non-secured PCs. Consumers, vendors and financial institutions need to know that the transactions, documents and identities are authentic. DES algorithm is the most used encryption method in data security for the Smartcard solutions.

1.3.5.2 AES Applications

AES can be used in any application that requires protection of data during transmission through the communication network, including applications such as electronic commerce transactions, ATM machines, wireless communication, Virtual Private Networks (VPN), and many others. Also it can be used as a part of the hardware or hybrid implementation of all major security protocols, including IPsec, SSL, IEEE 802.11a, and the ATM Forum Security Specification.

AES is now the industry standard for encryption. The National Security Agency (NSA) employs it for protecting secret information and industry uses the algorithm for creating commercially available encryption products.

File encryption and email encryption are two common applications for AES. File encryption protects the information on your hard disk or thumb drive. With encryption, your data will be secure even if your computer is hacked or your USB drive stolen. Email encryption protects your messages as they journey through the cloud and keeps them from being read by unintended recipients.

Thanks in large part to extensive input from the cryptographic community and the open review process, it can be trusted and is available to anyone who wishes to protect sensitive information.

1.3.5.3 IDEA Applications

Today, there are hundreds of IDEA-based security solutions available in many market areas, ranging from Financial Services, and Broadcasting to Government. IDEA is the name of a proven, secure, and universally applicable block encryption algorithm, which permits effective protection of transmitted and stored data against unauthorized access by third parties. The fundamental criteria for the development of IDEA were highest security requirements along with easy hardware and software implementation for fast execution.

The IDEA algorithm can easily be embedded in any encryption software. Data encryption can be used to protect data transmission and storage. Typical fields are:

- Audio and video data for cable TV, pay TV, video conferencing, distance learning, business TV, VoIP
- Sensitive financial and commercial data
- Email via public networks
- Transmission links via modem, router or ATM link, GSM technology
- Smart cards

1.4 Public-key Cryptography

The possibility of public key cryptography was first published in 1976 by Whitfield Diffie and Martin Hellman, who at the time were researchers at Stanford University. Ralph Merkle, a graduate student at the University of California, Berkeley, was studying the concept at the same time, but his ideas were not published until public key cryptography was well known. In their classic paper, Diffie and Hellman proposed the idea of public key cryptography and its use for exchanging keys, but not a public key cryptosystem. Several public key cryptosystems were subsequently proposed, but many were deemed insecure. Some systems are secure but are not practical for routine use either because the key is too large or because the ciphertext is significantly larger than the plaintext.

1.4.1 RSA

1.4.1.1 Overview

The RSA algorithm for public key cryptography, based on the idea that factorization of integers into their prime factors is hard to do, was proposed by (then) MIT professors Ronald Rivest, Adi Shamir, and Leonard Adleman in 1977. RSA has become one of the most successful algorithms for public key encryption and digital signatures. Many people had suspected that a government cryptographic agency such as the U.S. National Security Agency (NSA) had studied the possibility of public key encryption years earlier, but any evidence to this effect was classified. However, in 1997 CESG, a British cryptographic agency, released previously classified documents which revealed that James Ellis had discovered public key cryptography in 1970 and Clifford Cocks had internally published a version of the RSA algorithm in 1973. Nonetheless, Rivest, Shamir and Adleman are credited with the invention of RSA, and a patent for the algorithm was issued to MIT in 1983. The RSA patent will be discussed in more detail below.

A public key cryptosystem is made up of several components. There is a set of all possible plaintext messages, called M. There is also a set of keys, K. For each key $k \in K$, there is an encryption function $encrypt_k$ and a decryption function $decrypt_k$. These components must satisfy the following requirements:

1. $encrypt_k(decrypt_k(M)) = M$ and $decrypt_k(encrypt_k(M)) = M$ for every $m \in M$ and every $k \in K$.
2. For every M and every k, the values of $encrypt_k(M)$ and $decrypt_k(M)$ are easy to compute.
3. For almost every $k \in K$, if someone knows only the function $encrypt_k$, it is not computationally feasible to find an algorithm to compute $decrypt_k$.
4. Given $k \in K$, it is easy to find the functions $encrypt_k$ and $decrypt_k$.

1.4.1.2 Algorithm

As mentioned earlier, RSA is based on the idea that it is difficult to factor large numbers. This is what makes RSA secure, provided that the public key is sufficiently large. The following is a description of the mathematics of sending an encrypted message from Alice to Bob using the RSA algorithm:

1. Alice will choose two large (e.g. 512 or 1024-bit) prime numbers, P and Q.
2. Alice will choose an encryption key E such that E is less than the product $N = P \cdot Q$ and such that E and $(P-1) \cdot (Q-1)$ are relatively prime; in other words, $\gcd(E,(P-1) \cdot (Q-1)) = 1$. $(P-1) \cdot (Q-1)$ is referred to as $\phi(N)$, commonly called Euler's f function or Euler's Totient function. Also, gcd stands for greatest common divisor, which is defined as the largest factor that two numbers have in common.
3. Using the extended Euclidean algorithm, Alice will compute the decryption key D which has the property that $D \cdot E \equiv 1 \bmod \phi(N)$. This can also be written $D \equiv E^{-1} \bmod \phi(N)$. Mod is short for modulo; the modulo function over two variables, a and b, written a mod b is defined to be the remainder when a is divided by b.
4. The numbers P and Q are no longer needed and should be kept secret or discarded.
5. The numbers N and E are the public key and can be freely distributed. The numbers D and N are the private key and should be kept secret. Alice sends her public key to Bob.
6. Bob writes his message as a number M, which must be smaller than N. If M is larger than N, Bob breaks the message into blocks, each of which is less than N.
7. Bob calculates the ciphertext $C = \text{encrypt}(M) \equiv M^E \bmod N$ and sends C to Alice.
8. Alice receives the ciphertext C and decrypts it to find the original plaintext message using the function $M = \text{decrypt}(C) \equiv C^D \bmod N$.
9. Note that the encryption and decryption functions can be "reversed" i.e., Alice could have encrypted a message M using her private key D. She could then send the encrypted message C to Bob, who would use Alice's public key E to decrypt C.

Table 3 summarizes this method.

Table 3: *RSA Algorithm Summary*

Public Key	N = P·Q (where P and Q are prime numbers that are kept secret E where gcd(E,ϕ(N)) = 1
Private Key	$D \equiv E^{-1} \bmod \phi(N)$
Encryption Function	$\text{encrypt}(M) \equiv M^E \bmod N = C$
Decryption Function	$\text{decrypt}(M) \equiv C^D \bmod N = M$

1.4.1.3 Breaking RSA

The strength and security of the RSA algorithm rely on the difficulty of factoring large numbers. It should be noted, however, that it has not been mathematically proven that the only way to determine the plaintext message M from the ciphertext and the public key is by factoring N. It is theoretically possible that some entirely new method will be devised to find M. If such a method were discovered, it could also be used as a factoring method, and since this mathematical problem has been studied for hundreds of years, the mathematical community is confident that the RSA algorithm is quite secure. In fact, RSA has withstood years of extensive cryptanalysis.

Factoring N is possible if the key length is small enough and if enough computing resources can be devoted to the task. There are a number of different factoring methods than can be employed, such as the Universal Exponent Factorization Method, the Exponent Factorization Method, Pollard's p – 1 Factoring Algorithm, the Quadratic Sieve, and the Number Field Sieve. The Number Field Sieve was successfully used in 1999 to factor both a 140- and a 512-bit RSA key and, at present, is the most powerful factoring method known.

RSA Laboratories publishes a series of cryptographic challenges to the public. The goals of the RSA Factoring Challenges are to help to encourage research into computational number theory and factoring techniques and to assure the public of RSA's security. Cash prizes are awarded to successful participants, and the results of these challenges are available to the public and are used to help RSA users determine suitable key lengths for various levels of security. When referring to RSA key lengths, one is actually referring to the size of N, the product of the primes P and Q. Thus if P and Q are both 256-bit numbers, then N is a 512-bit RSA key. In fact, a 515-bit N is a 155-digit number. Factoring the 155-digit RSA Challenge number was accomplished on August 22, 1999, by an international group of researchers

using computers located in 11 different sites around the world. The team required 5.2 months, plus an additional nine weeks for preliminary computations, to factor RSA-155. This translated to about 35.7 CPU years. In contrast, it took only 8.9 CPU years and 9 weeks of calendar time to factor RSA-140, the 140-bit RSA Challenge number. Increasing the key size dramatically increases the difficulty of the resulting factoring problem. As a result of the RSA Factoring Challenges, as well as other research, the current minimum recommended key size for RSA is 768 bits. Key sizes of 1024 bits or even 2048 bits are not uncommon. When choosing a key size one needs to consider a number of factors, including the importance of the data, how long the data will need to remain secure, and the resources available to an adversary. For example, a much larger RSA key would be used to protect nuclear secrets than would be used to protect routine email messages. Computing power will continue to improve, and factoring methods have made great strides and presumably will continue to do so, but these methods are still very slow. The RSA algorithm, with a sufficiently large key length, remains highly secure.

A method of attacking RSA is through the use of timing attacks. This method was discovered in 1995 by Paul Kocher while he was an undergraduate student at Stanford University. Using the fact that many implementations of cryptography do things at different speeds for different keys, he demonstrated that it is possible to determine the private key being used by taking careful measurements of the length of time it takes to accomplish a series of decryptions.

In 1998, Daniel Bleichenbacher described the first practical adaptive chosen ciphertext attack, against RSA-encrypted messages using the PKCS[5] #1 v1 padding scheme (a padding scheme randomizes and adds structure to an RSA-encrypted message, so it is possible to determine whether a decrypted message is valid). Due to flaws with the PKCS #1 scheme, Bleichenbacher was able to mount a practical attack against RSA implementations of the Secure Socket Layer protocol, and to recover session keys. As a result of this work, cryptographers now recommend the use of provably secure padding schemes such as Optimal Asymmetric Encryption Padding, and RSA Laboratories has released new versions of PKCS #1 that are not vulnerable to these attacks.

[5] In cryptography, PKCS is a group of public-key cryptography standards devised and published by RSA Security Inc, starting in the early 1990s

1.4.2 Applications

1.4.2.1 RSA Applications

The RSA system is currently used in a wide variety of products, platforms, and industries around the world. It is found in many commercial software products and is planned to be in many more. The RSA algorithm is built into current operating systems by Microsoft, Apple, Sun, and Novell. In hardware, the RSA algorithm can be found in secure telephones, on Ethernet network cards, and on smart cards. In addition, the algorithm is incorporated into all of the major protocols for secure Internet communications, including S/MIME, SSL, and S/WAN. It is also used internally in many institutions, including branches of the U.S. government, major corporations, national laboratories, and universities.

The RSA public-key cryptosystem can be used to authenticate or identify another person or entity. The reason it works well is because each entity has an associated private key which (theoretically) no one else has access to. This allows for positive and unique identification.

RSA can be used to construct signature schemes. The signature function corresponds to the decryption function parameterized by the user's secret key and the verification function is derived from the encryption function. Thus in the RSA signature scheme for example, a user signs a message m by applying the RSA decryption function to his secret key d. To verify the signature, it suffices to apply the RSA encryption function (parameterized by the associated public key (e, n)) and to verify that the result of this calculation does indeed correspond to the clear text sent.

When one wants to ensure the confidentiality of exchanged messages, one does not in general have access to a single type of cipher system. In effect, the complexity of the operations involved in public-key systems renders the cipher system extremely slow compared to a secret-key system. On the other hand, only a public-key scheme allows a secure exchange of a secret without preliminary exchange of a shared secret. Thus one would prefer to use a public-key algorithm to exchange a secret key. This key will serve to encrypt the exchange of information with the aid of a symmetric algorithm. This combination of the two techniques permits both the speed of secret-key encryption and the resolution of the problem of exchanging secret keys between the two interlocutors. This is

notably the encryption solution used in the PGP program. More generally, public-key systems are used in practice to encrypt very short messages.

Finally, RSA is used on bank cards. When one uses a bank card to pay for a small purchase, the operation is done off-line, without exchanging any information with the bank (bank information is assembled and communicated at the end of the day). The unique control at the moment of payment (besides the confidential code), consists in verifying that the card being used is valid and this procedure is done using an RSA signature. Each card has an identifier which has been signed by the bank. It is this signature, written on the chip, which is verified at each transaction by the business terminal. Each card bearing a valid signature is therefore considered authentic since the bank is the only authority which has the RSA secret key allowing the signing.

2 Crypto Architecture

2.1 Introduction

Every IP consists of components combined using a certain architecture. The main functionality of the IP is implemented by the main engines (described in section 2.2.4), but except of these engines an interface should exist in order to communicate with a main CPU. One of the most popular communication interfaces is the AMBA AHB interface (described in section 2.2.1). The module which is responsible to implement this interface is commonly called Main Controller (described in section 2.2.2). The communication between the user and the IP is carried out using certain memory mapped registers accessible by the user with their unique address. All these registers are placed in a module called Register File (described in section 2.2.3).

The aforementioned components are combined together as depicted in Figure 24.

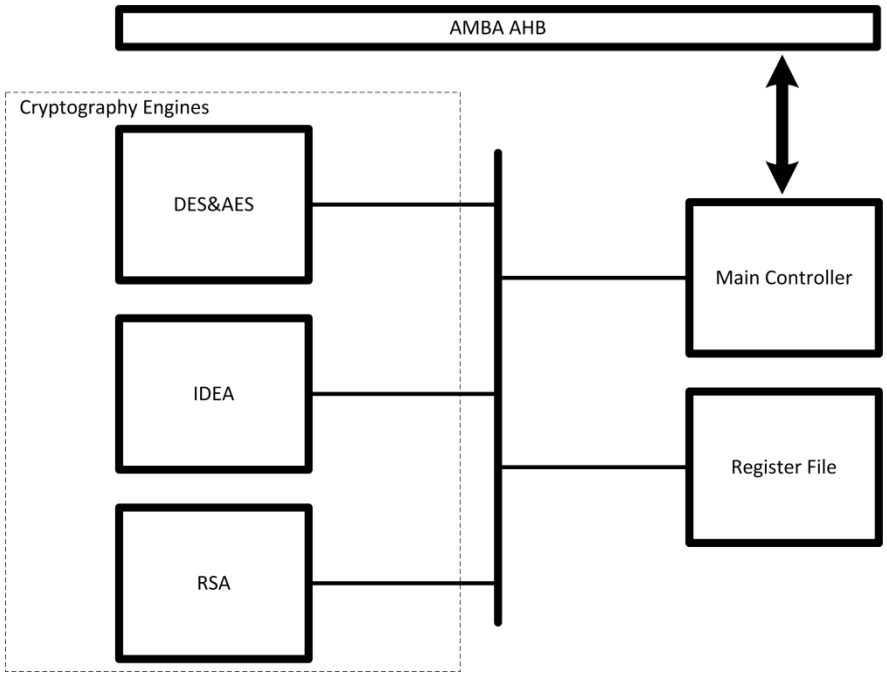


Figure 24: *Crypto IP Architecture Overview*

2.2 Crypto Components

2.2.1 AMBA AHB Interface

2.2.1.1 Introduction

In this section an on-chip communication standard is described, called Advanced Microcontroller Bus Architecture (AMBA). AMBA specification defines three distinct bus architectures:

- The advanced high-performance bus (AHB) for high clock frequency modules
- The advanced-system bus (ASB) also for high clock frequency modules
- The advanced peripheral bus (APB), which is mainly used for low-power peripheral modules

The AHB bus is used as the backbone bus for high-performance systems and supports connection between embedded processor, on-chip memories and off-chip memory interfaces or bridges to low-performance system where most of the peripheral devices located. AHB bus is more complex and has more high-performance features than ASB bus, which is the alternative choice for system bus. APB is optimized for minimal power consumption and reduces complexity to peripheral device integration. APB is usually used for interfacing peripheral devices with low bandwidth.

Features of each bus architecture are mentioned in Table 4.

Table 4: *Features of different AMBA Buses*

AMBA AHB	AMBA ASB	AMBA APB
High performance	High performance	Low Power
Pipelined operation	Pipelined operation	Latched address and control
Multiple bus masters	Multiple bus masters	Simple interface
Burst transfers		Suitable for many peripherals
Split transactions		

An AMBA-based microcontroller (see Figure 25: *Typical AMBA System*) typically consists of a high-performance system backbone bus (AMBA AHB or AMBA ASB), able to sustain the

external memory bandwidth, on which the CPU, on-chip memory and other Direct Memory Access (DMA) devices reside. This bus provides a high-bandwidth interface between the elements that are involved in the majority of transfers. Also located on the high performance bus is a bridge to the lower bandwidth APB, where most of the peripheral devices in the system are placed. While transferring data from the system's processor to peripheral devices like UART, timer, peripheral I/O and keyboard, the bridge converts the transferred signals from one type to another, to satisfy different performance and protocol requirements.

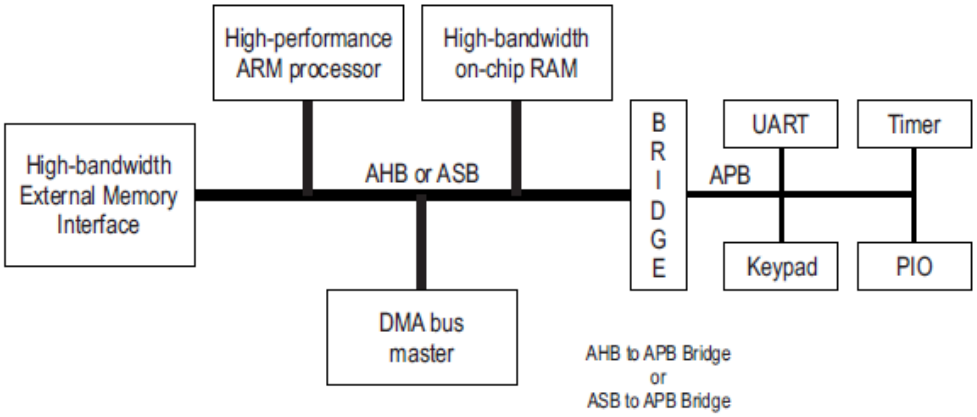


Figure 25: *Typical AMBA System*

2.2.1.2 AMBA AHB Overview

AHB is a flavor of AMBA bus which is intended to address the requirements of high-performance synthesizable embedded designs. It is a high-performance system bus that supports multiple bus masters and provides high-bandwidth operation. AMBA AHB implements the features required for high-performance, high clock frequency systems including:

- burst transfers
- split transactions
- single-cycle bus master handover
- single-clock edge operation
- non-tristate implementation
- wider data bus configurations (64/128 bits)

Bridging between this higher level of bus and the current ASB/APB can be done efficiently to ensure that any existing designs can be easily integrated. An AMBA AHB design may contain one or more bus masters, typically a system would contain at least the processor and test interface. However, it would also be common for a Direct Memory Access (DMA) or Digital Signal Processor (DSP) to be included as bus masters.

The external memory interface, APB bridge and any internal memory are the most common AHB slaves. Any other peripheral in the system could also be included as an AHB slave. However, low-bandwidth peripherals typically reside on the APB. A typical AMBA AHB system design contains the following components:

- AHB master: A bus master is able to initiate read and write operations by providing an address and control information. Only one bus master is allowed to actively use the bus at any one time.
- AHB slave: A bus slave responds to a read or write operation within a given address-space range. The bus slave signals back to the active master the success, failure or waiting of the data transfer.
- AHB arbiter: The bus arbiter ensures that only one bus master at a time is allowed to initiate data transfers. Even though the arbitration protocol is fixed, any arbitration algorithm, such as highest priority or fair access can be implemented depending on the application requirements. An AHB would include only one arbiter, although this would be trivial in single bus master systems.
- AHB decoder: The AHB decoder is used to decode the address of each transfer and provide a select signal for the slave that is involved in the transfer. A single centralized decoder is required in all AHB implementations.

2.2.1.3 AMBA AHB Signal List

This section contains an overview of the AMBA AHB signals (see Table 5: *AMBA AHB Signals*). All signals are prefixed with the letter H, ensuring that the AHB signals are differentiated from other similarly named signals in a system design.

Table 5: *AMBA AHB Signals*

Name	Source	Description
HCLK *Bus clock*	Clock source	This clock times all bus transfers. All signal timings are related to the rising edge of HCLK.
HRESETn *Reset*	Reset controller	The bus reset signal is active LOW and is used to reset the system and the bus. This is the only active LOW signal.
HADDR[31:0] *Address bus*	Master	The 32-bit system address bus.
HTRANS[1:0] *Transfer type*	Master	Indicates the type of the current transfer, which can be NONSEQUENTIAL, SEQUENTIAL, IDLE or BUSY.
HWRITE *Transfer direction*	Master	When HIGH this signal indicates a write transfer and when LOW a read transfer.
HSIZE[2:0] *Transfer size*	Master	Indicates the size of the transfer, which is typically byte (8-bit), halfword (16-bit) or word (32-bit). The protocol allows for larger transfer sizes up to a maximum of 1024 bits.
HBURST[2:0] *Burst type*	Master	Indicates if the transfer forms part of a burst. Four, eight and sixteen beat bursts are supported and the burst may be either incrementing or wrapping.
HPROT[3:0] *Protection control*	Master	The protection control signals provide additional information about a bus access and are primarily intended for use by any module that wishes to implement some level of protection. The signals indicate if the transfer is an opcode fetch or data access, as well as if the transfer is a privileged mode access or user mode access. For bus masters with a memory management unit these signals also indicate whether the current access is cacheable or bufferable.
HWDATA[31:0] *Write data bus*	Master	The write data bus is used to transfer data from the master to the bus slaves during write operations. A minimum data bus width of 32 bits is recommended. However, this may easily be extended to allow for higher bandwidth operation.
HSELx *Slave select*	Decoder	Each AHB slave has its own slave select signal and this signal indicates that the current transfer is intended for the selected slave. This signal is simply a combinatorial decode of the address bus.
HRDATA[31:0] *Read data bus*	Slave	The read data bus is used to transfer data from bus slaves to the bus master during read operations. A minimum data bus width of 32 bits is recommended. However, this may easily be extended to allow for higher bandwidth operation.

HREADY *Transfer done*	Slave	When HIGH the HREADY signal indicates that a transfer has finished on the bus. This signal may be driven LOW to extend a transfer. Note: Slaves on the bus require HREADY as both an input and an output signal.
HRESP *Transfer response*	Slave	The transfer response provides additional information on the status of a transfer. Two different responses are provided, OKAY and ERROR.

2.2.1.4 Bus Interconnection

The AMBA AHB bus protocol is designed to be used with a central multiplexor interconnection scheme. Using this scheme all bus masters drive out the address and control signals indicating the transfer they wish to perform and the arbiter determines which master has its address and control signals routed to all of the slaves. A central decoder is also required to control the read data and response signal multiplexor, which selects the appropriate signals from the slave that is involved in the transfer. Figure 26 illustrates the structure required to implement an AMBA AHB design with three masters and four slaves.

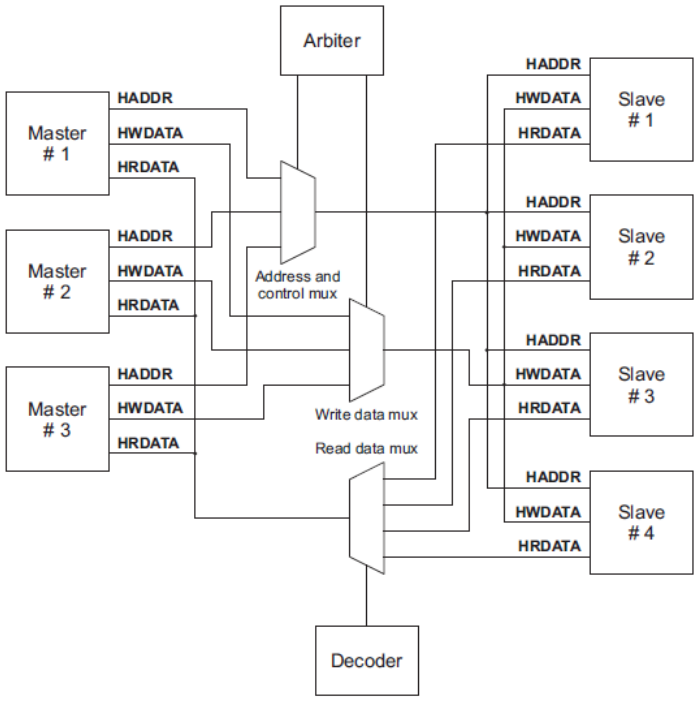


Figure 26: *AMBA AHB Interconnection*

2.2.1.5 AMBA AHB Operation

Before an AMBA AHB transfer can commence the bus master must be granted access to the bus. This process is started by the master asserting a request signal to the arbiter. Then the arbiter indicates when the master will be granted use of the bus. A granted bus master starts an AMBA AHB transfer by driving the address and control signals. These signals provide information on the address, direction and width of the transfer, as well as an indication if the transfer forms part of a burst.

A write data bus is used to move data from the master to a slave, while a read data bus is used to move data from a slave to the master. An AHB transfer consists of two distinct sections:

- The address phase, which lasts only a single cycle.
- The data phase, which may require several cycles.

2.2.1.6 AHB Bus Slave

An AHB bus slave responds to transfers initiated by bus masters within the system. The slave uses a HSELx select signal from the decoder to determine when it should respond to a bus transfer. All other signals required for the transfer, such as the address and control information, will be generated by the bus master. The interface of a bus slave is depicted in Figure 27.

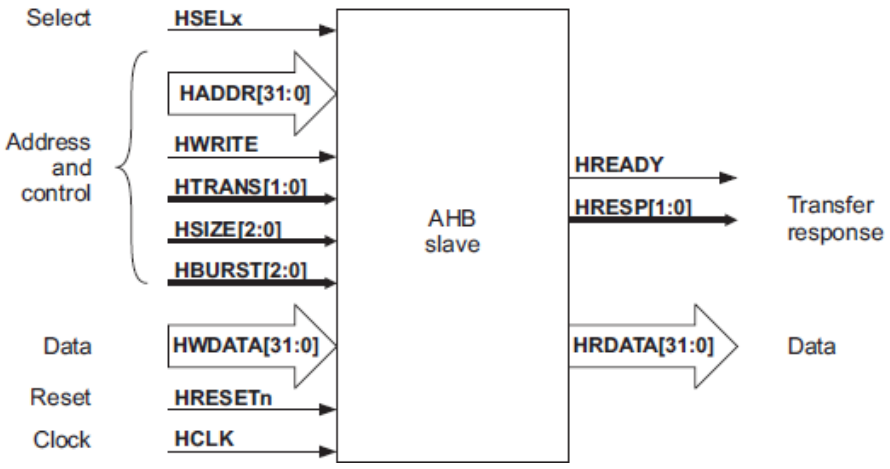


Figure 27: *AMBA AHB Slave*

2.2.1.7 Implementation

The Crypto IP is in fact an AMBA AHB slave and in order to implement the signals of an AMBA AHB interface (as shown in Figure 27) a module called “Main Controller” described in section 2.2.2 has been created.

2.2.2 Main Controller

2.2.2.1 Implementation

The main controller is the module which implements the AMBA AHB interface, communicates with the register file (described in section 2.2.3) and feeds the cryptographic engines of the IP (described in section 2.2.4) in order to perform certain operations.

In order to meet the conditions of the AMBA AHB there is an FSM (see Figure 28) which separates the address phase from the data phase, checks the AMBA AHB inputs, and exports the proper AMBA AHB outputs.

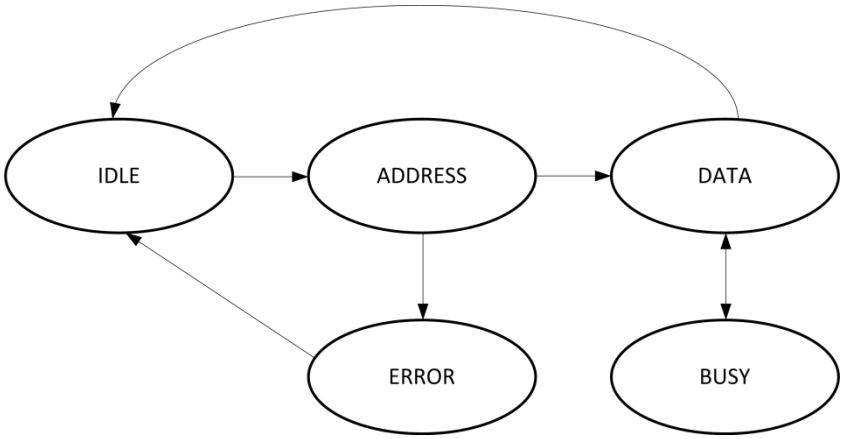


Figure 28: *Main Controller FSM*

The user gives the proper values to the main controller through the AMBA AHB interface and the controller promotes them to the register file. All registers in the register file are 32-bit long, but some engines have inputs of bigger size (multiple of 32). As a result, inside the

main controller, there are some big registers which combine the 32-bit fragments from the register file (by shifting the existing value 32 times and place each new fragment in the least significant bits) in order to create the input passed to the cryptographic engines.

Using these big registers as well as other signals from the register file, the main controller feeds the cryptographic engines with the proper inputs and collects the results in order to be placed in the register file so that the user can read them using the AMBA AHB interface.

Figure 29 illustrates the block diagram of the main controller. In this block diagram the input processor, the register file and the cryptographic engines are illustrated. The input processor module implements the AMBA AHB interface and the register file module contains the registers mentioned in section 2.2.3.

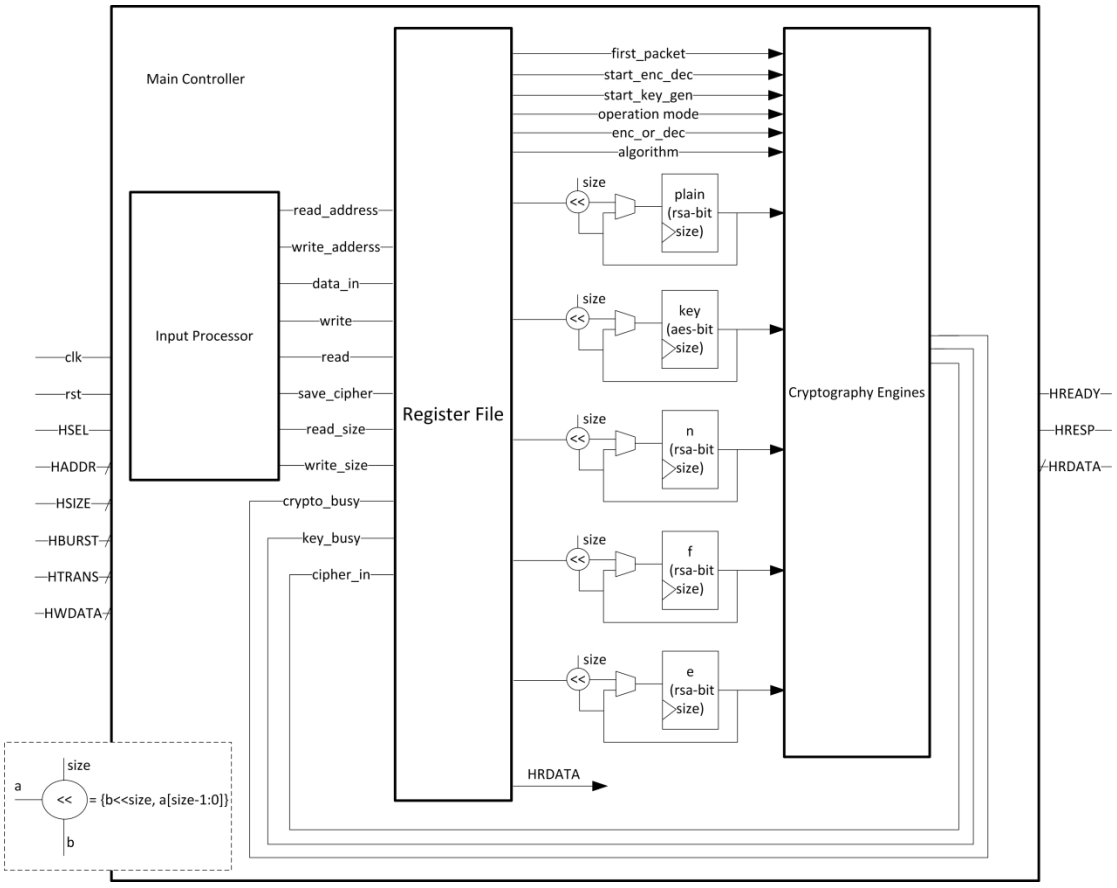


Figure 29: *Main Controller Block Diagram*

The main controller is the module which interacts with the user in order to perform the process of key generation and encryption/decryption.

2.2.2.2 Process of Key Generation and Encryption / Decryption

The following sequence of steps has to be executed to perform a key generation:

1) Write the key in 32-bit fragments (as many as needed) in the *crypto_key_reg* starting from the most significant word
2) Wait until *key_busy* and *crypto_busy* (in the *crypto_status_reg*) are LOW
3) Set the proper bits of *crypto_ctrl_reg* (*start_key_gen*, *cipher_sel*)

The following sequence of steps has to be executed to perform an encryption/decryption:

1) Write the plain text in 32-bit fragments (as many as needed) in the *crypto_in_reg* starting from the most significant word
2) Wait until *key_busy* and *crypto_busy* (in the *crypto_status_reg*) are LOW
3) Set the proper bits of *crypto_ctrl_reg* (*start_enc_dec*, enc_dec, *cipher_sel etc.*)
4) Wait until *crypto_busy* (in the *crypto_status_reg*) is LOW to read the output.

Note that no step can be executed before the previous one is finished and that in order to read the output you have to read as many 32-bit fragments needed from the *crypto_out_reg* (least significant words come first).

In Figure 30 and Figure 31 there is a waveform where a key generation and an encryption process are performed.

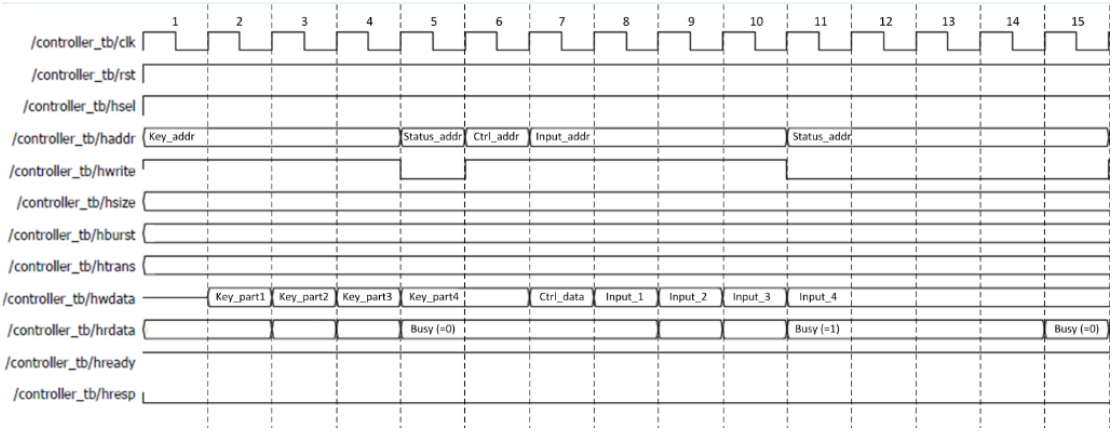


Figure 30: *Waveform Part 1*

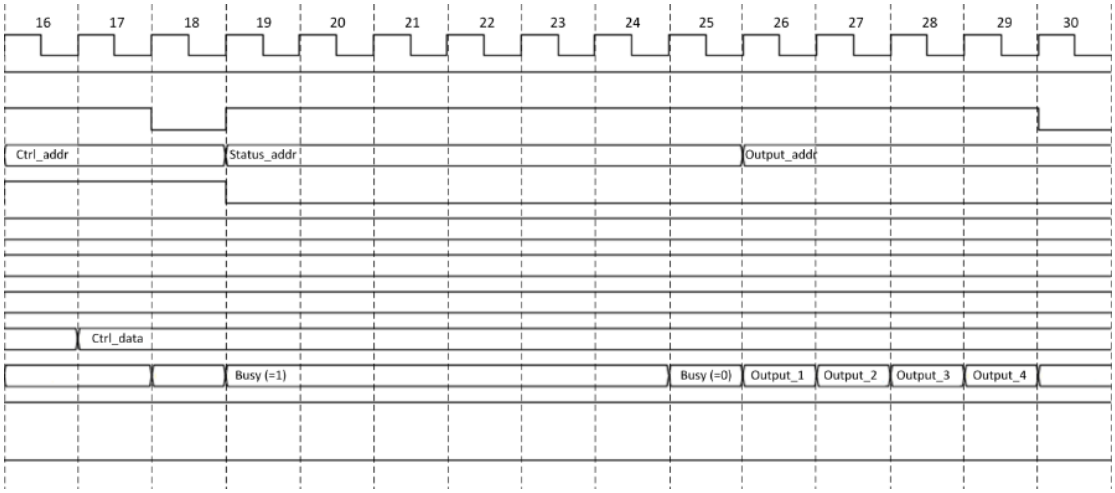


Figure 31: *Waveform Part 2*

The sub-processes executed in the waveform are described below:

- Cycles 1-5: key transfer
- Cycle 5: check if busy (read *key_busy*, *crypto_busy*)
- Cycles 6-7: set *control_reg* to start key generation
- Cycles 7-11: plain transfer
- Cycles 11-15: check if busy (read *key_busy*, *crypto_busy*)
- Cycles 16-17: set *control_reg* to start encryption
- Cycles 19-25: check if busy (read *crypto_busy*)
- Cycles 26-29: read the cipher

2.2.3 Register File

The register file is an array of the input and output registers of an IP and is part of the architecture visible to the programmer. Read and write operations can be performed to these registers as each one of them has a specific address.

The register file implemented for the Crypto IP consists of the following registers:

- Control Register (Name: *crypto_ctrl_reg*, Address: 0x0000)

Table 6: *Control Register Specification*

Bit	Symbol	Description	Mode	Reset Value
31-12	-	RESERVED	-	0
11	FIRST PACKET	If this packet is the first one of the stream 0: Not the first packet 1: The first packet This bit is autocleared after one cycle	R/W	0
10	START_ENC_DEC	Start encryption/decryption of plain message 0: Keep idle 1: Start the encryption/decryption This bit is autocleared after one cycle	R/W	0
9	START_KEY_GEN	Start key generation if such configuration exists. 0: Keep idle 1: Start the key generation This bit is autocleared after one cycle	R/W	0
8-4	BC_MODE	Mode of operation for DES&AES 0: ECB 1: CBC 2: PCBC 3: CFB 4: OFB 5: CTR 6-31: Reserved	R/W	0
3	ENC_DEC	Selects whether encryption or decryption is to be performed 0: Encryption 1: Decryption	R/W	0
2-0	CIPHER_SEL	Selection of the algorithm to be used. Enables the respective engine in the IP. 0: RSA 1: AES 2: DES 3: IDEA 4-7: Reserved	R/W	0

- Status Register (Name: *crypto_status_reg*, Address: 0x0004)

Table 7: *Status Register Specification*

Bit	Symbol	Description	Mode	Reset Value
31-2	-	RESERVED	-	0
1	KEY_BUSY	Notifies that the system is busy with key generation tasks 0: Idle 1: Busy	R	0
0	CRYPTO_BUSY	Notifies that the system is busy with encryption/decryption tasks 0: Idle 1: Busy	R	0

- Input Register (Name: *crypto_in_reg*, Address: 0x0008)

Table 8: *Input Register Specification*

Bit	Symbol	Description	Mode	Reset Value
31-0	INPUT	Contains a plain text word if encryption is to be performed or a cipher's word if decryption is selected	R/W	0

- Key Register (Name: *crypto_key_reg*, Address: 0x000C)

Table 9: *Key Register Specification*

Bit	Symbol	Description	Mode	Reset Value
31-0	KEY	Contains a Key word	R/W	0

- RSA_N Register (Name: *crypto_rsa_n_reg*, Address: 0x0010)

Table 10: *RSA_N Register Specification*

Bit	Symbol	Description	Mode	Reset Value
31-0	RSA_N	Contains a word of the product p·q used for RSA	R/W	0

- RSA_F Register (Name: *crypto_rsa_f_reg*, Address: 0x0014)

Table 11: *RSA_F Register Specification*

Bit	Symbol	Description	Mode	Reset Value
31-0	RSA_F	Contains a word of the product (p-1)·(q-1) used for RSA	R/W	0

- RSA_E Register (Name: *crypto_rsa_e_reg*, Address: 0x0018)

Table 12: *RSA_E Register Specification*

Bit	Symbol	Description	Mode	Reset Value
31-0	RSA_E	Contains a word of the public key exponent used for RSA	R/W	0

- Output Register (Name: *crypto_out_reg*, Address: 0x001C)

Table 13: *Output Register Specification*

Bit	Symbol	Description	Mode	Reset Value
31-0	OUTPUT	Contains a word of the resulting cipher/plain text if encryption/decryption has been performed respectively	R	0

In Figure 32, the block diagram of the register file is depicted, where all the registers mentioned above are included.

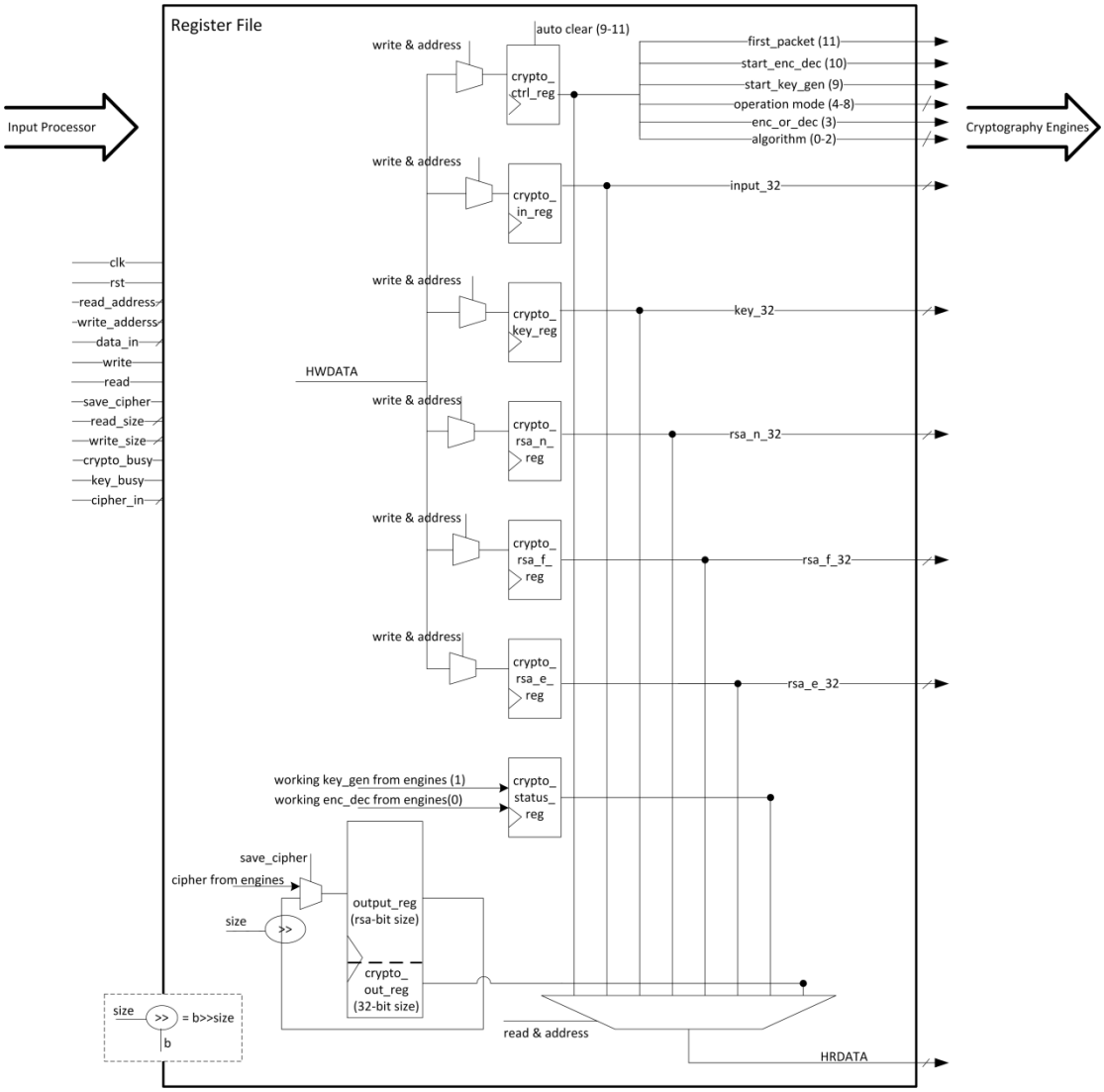


Figure 32: *Register File Block Diagram*

As shown in the block diagram, there is one register called “output” in the register file and its least significant 32 bits are called *crypto_out_reg*. This register operates as a shift register for the encryption/decryption result. After an encryption/decryption process, the result is stored in this output register and each time that the user reads x bits from the *crypto_out_reg*, the output register shifts x times right.

2.2.4 Cryptography Engines

The main function of the Crypto IP is to perform encryption/decryption processes using certain algorithms. Four encryption algorithms are implemented:

- Data Encryption Standard (DES)
- Advanced Encryption Standard (AES)
- International Data Encryption Algorithm (IDEA)
- RSA

DES and AES are implemented in a single engine called DES&AES engine, which also implements the block cipher operation modes. IDEA and RSA have their own engines called IDEA engine and RSA engine respectively. These three engines are described in the following chapters.

Main Controller (described in section 2.2.2) receives user’s input, selects which engine will be used, feeds it with the proper inputs and collects its outputs.

3 DES & AES Engine

3.1 Introduction

In this chapter, one of the hardware accelerators used in this IP is presented, which implements the DES and AES algorithms (mentioned in sections 1.3.1 and 1.3.2) as well as the block cipher operation modes (mentioned in section 1.3.4). In section 3.2 the compile time parameters of the DES&AES engine which can be configured by the user to modify the engine according to the specifications and requirements are displayed. In section 3.3 the details of the algorithms' implementation are given. The main parameters considered during the implementation were the area and the frequency of the engine. In section 3.4 the implementation's results in ASIC technologies are illustrated.

3.2 Configuration Parameters

There are five compile time configuration parameters in the DES&AES engine which are explained below:

- des_version: this parameter defines which version of the DES module will be used, the speed optimized or the area optimized.
- aes_version: this parameter defines which version of the AES module will be used, the speed optimized or the area optimized.
- use_des_key_generator: this parameters defines whether the DES key generator module will be used or not.
- use_aes_key_generator: this parameters defines whether the AES key generator module will be used or not.
- aes_key_size: this parameter defines the bit size of the key used in AES module. The valid values of this parameter are 128/192/256.

3.3 Implementation

3.3.1 General Description

There are two versions implementing the DES algorithm, the speed optimized (SO) and the area optimized (AO). The same two versions exist for the AES algorithm. The basic features of the DES&AES engine are the following:

- AES & DES engine implements hardware data encryption and decryption using AES and DES encryption modules
- DES processes 64-bit data blocks with 64-bit key
- AES processes 128-bit data blocks with 128/192/256-bit key
- Fully synchronous design
- Encryption and decryption unit in single core
- For both AES and DES, two versions are available for the user to select:
    - Area Optimized version (**small area/resources** utilization)
    - Speed Optimized version (fully **pipelined**)
- Key generator modules included for both AES and DES
- Key generator can be ignored using ready keys inserted from the user
- External memory not required
- All basic modes of operation available (EBC, CBC, PCBC, CFB, OFB, CTR)
- Available signals to indicate when input data can be inserted an when the output is ready

3.3.2 Pin Description

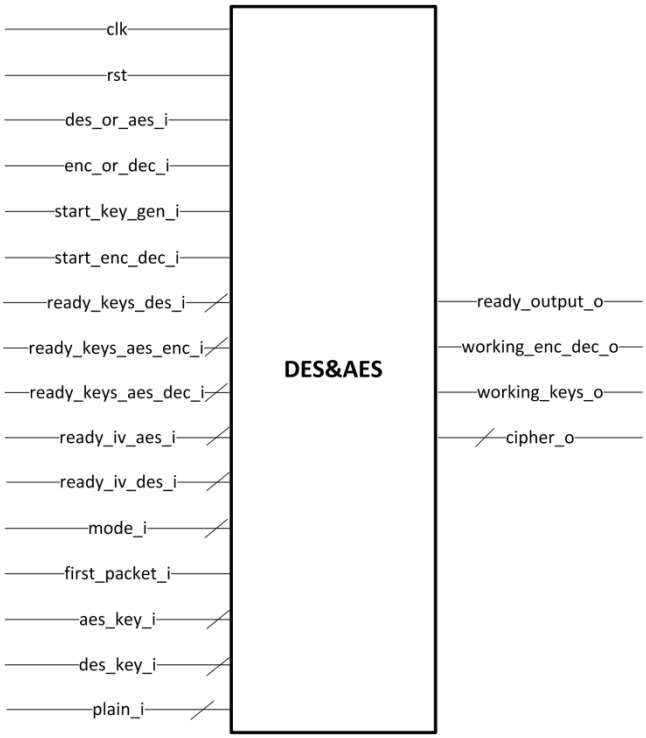


Figure 33: *DES&AES Engine Symbol Diagram*

Table 14 contains the description of each input/output pin existing in the DES&AES engine.

Table 14: *DES&AES Engine Pin Description*

Name	Type	Width (bits)	Description
clk	input	-	clock signal
rst	input	1	reset signal (resets when LOW)
des_or_aes_i	input	1	select engine (LOW for DES / HIGH for AES)
mode_i	input	4	select mode of operation (1)
enc_or_dec_i	input	1	LOW for encryption / HIGH for decryption
start_key_gen_i	input	1	HIGH to start a new key generation
first_packet_i	input	1	HIGH for the first packet of the stream
plain_i	input	128 (2)	input data
aes_key_i	input	128/192/256	AES encryption key

des_key_i	input	64	DES encryption key
start_enc_dec_i	input	1	HIGH to start a new encryption / decryption
ready_keys_des_i	input	768	ready DES keys if key generator not used
ready_keys_aes_enc_i	input	1408/1664/1920	ready AES encryption keys if key generator not used
ready_keys_aes_dec_i	input	1408/1664/1920	ready AES decryption keys if key generator not used
ready_iv_des_i	input	64	ready DES block cipher modes initialization vector if key generator not used
ready_iv_aes_i	input	128	ready AES block cipher modes initialization vector if key generator not used
cipher_o	output	128 (2)	output data
ready_output_o	output	1	HIGH when output is ready
working_enc_dec_o	output	1	LOW when encryption / decryption finished
working_keys_o	output	1	LOW when key generation finished

Notes

1) Operation modes

bits	mode of operation
0000	ECB
0001	CBC
0010	PCBC
0011	CFB
0100	OFB
0101	CTR

2) The 64 least significant bits are used when DES is selected

3.3.3 Process of Key Generation and Encryption / Decryption

The following sequence of steps has to be executed to perform a key generation:

1) Wait until *working_enc_dec_o* and *working_keys_o* are LOW
2) Set *des_or_aes_i* to select the encryption algorithm used and *aes_key_i* or *des_key_i* (depends on the selected algorithm) to provide the key
3) Set *start_key_gen_i* (pulse) to start the key generation process

The following sequence of steps has to be executed to perform an encryption/decryption:

1) Wait until *working_enc_dec_o* and *working_keys_o* are LOW
2) Set *des_or_aes_i* to select the encryption algorithm used, *mode_i* to select the mode of operation, *enc_or_dec_i* to select between encryption and decryption and *plain_i*
3) Set *start_enc_dec_i* (pulse) to start the encryption/decryption process
4) Wait until *ready_output_o* is HIGH to read the result of the process

Note that no step can be executed before the previous one is finished. Also, if the key generator is not used, there are no signals aes*_key_i*, *des_key_i*, *working_keys_o* and the initialization vector must be declared by setting the signals *ready_iv_des_i*, *ready_iv_aes_i* (depends on the selected algorithm). The key update is performed by setting the signals *ready_keys_des_i, ready_keys_aes_enc_i, ready_keys_aes_dec_i, des_or_aes_i* (depends on the selected algorithm) and then setting the signal *start_key_gen_i* (pulse) when *working_enc_dec_o* is LOW.

3.3.4 Algorithmic Details

3.3.4.1 AES Encryption Process

As mentioned in section 1.3.2.4 in each encryption round the following steps are executed:

1. SubBytes
2. ShiftRows
3. MixColumns
4. AddRoundKey

For the MixColumns step the aforementioned Table 1: *MixColumns Multiplication Matrix* is used. Each word (a) is multiplied with the matrix and the result's form (r) is the following:

- $r_i = x \cdot a_0 \oplus y \cdot a_1 \oplus z \cdot a_2 \oplus w \cdot a_3$ (where a_i are the bytes of a)

To compute the above multiplications, Table 48: *Rijndael N-Box* and Table 49: *Rijndael E-Box* are used as shown below:

- $a_i \cdot b$ = E-Box{ (N-Box(a) + N-Box(b)) (mod 0xFF) }

However, the MixColumns matrix's elements have only 3 different values. So N-Box(b) is a constant with 3 different values and the relation becomes the following:

- $a \cdot b$ = E-Box{ (N-Box(a) + cnst) (mod 0xFF) }

The SubBytes step is performed using Table 47: *Rijndael S-Box* and it is obvious that it can be reordered with the Shift Rows step. Hence, after shifting the initial word and before the final XOR, each byte follows the procedure depicted in Figure 34.

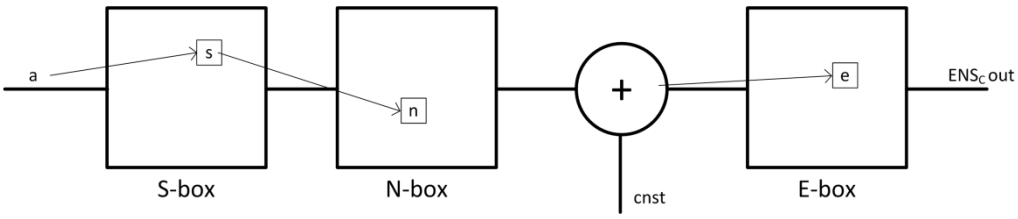


Figure 34: *ENS Operation*

The above operations can be combined to a single box called ENS-Box, but as the ENS-Box depends on the constant, there are three different ENS-Boxes called ENS$_C$-Box where c is the constant (see Table 52: *ENS$_2$-Box* and Table 53: *ENS$_3$-Box*). It is obvious that:

- ENS$_1$-Box = S-Box

So by combining those three boxes to a single one the encryption procedure becomes quite easier. The final procedure is depicted in Figure 35.

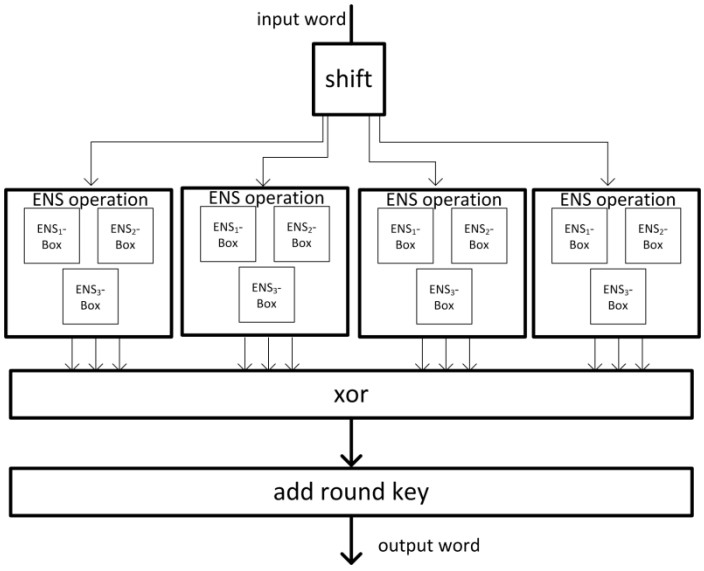


Figure 35: *Modified AES Main Process*

3.3.4.2 AES Decryption Process

A similar procedure takes place in each decryption round but the order of the steps and the tables used are different. The decryption steps are the following:

1. InvShiftRows
2. InvSubBytes
3. AddRoundKey
4. InvMixColumns

The InvShiftRows step is the same as the ShiftRows step but instead of performing a left rotation, a right one is performed. The InvSubBytes step is the same as the SubBytes step using a different box (see Table 50: *Rijndael Inverse S-Box*) and the InvMixColumns step is the same as the MixColumns step using a different matrix (see Table 51: *InvMixColumns Multiplication Matrix*).

If the output of step 2 is called *state* then the output of all four steps is the following:

- InvMixColumns (state $\oplus$ key)

But as these operations are linear, the above relation can be transformed as shown below:

- InvMixColumns (state) $\oplus$ InvMixColumns(key)

So the InvMixColumns step can be executed in step 3 and the AddRoundKey in step 4, to follow the order of the encryption.

The only difference in the AddRoundKey step is that instead of using the decryption keys

- $dec\_key_i = enc\_key_{N-i}$ (where N is the number of AES rounds)

the InvMixColumns(dec_key_i) is used.

As a result, the decryption is performed exactly as the encryption using the above decryption keys and the four new inverse ENS_C-Boxes generated (see Table 54: *Inverse ENS_E-Box*, Table 55: *Inverse ENS_B-Box*, Table 56: *Inverse ENS_D-Box* and Table 57: *Inverse ENS_9-Box*), as the InvMixColumns matrix has different constants from the MixColumns matrix and the InvSubBytes uses a different S-Box. Also in order to perform the InvMixColumns operation in the decryption keys, four new tables are generated. Each one of them combines the E-Box and N-Box for a different constant of the InvMixColumns matrix in the same way as described above (see Table 58: *EN_E-Box*, Table 59: *EN_B-Box*, Table 60: *EN_D-Box* and Table 61: *EN_9-Box*).

3.3.5 Implementation Details

There are two different implementations of the DES and AES algorithms, as mentioned in section 3.3.1, the speed optimized (SO) and the area optimized (AO). The way these versions are implemented is the same in both algorithms. Each algorithm has a main encryption round which is executed more than one times depending on the algorithm. In the SO versions the sub-module of the encryption round is generated as many times required and these sub-modules are wired and operate in one cycle. In the AO versions there is only one instance of the encryption round which is reused.

Another worth mentioning implementation detail is the one of the operation modes described in section 1.3.4. All modes have in common that in each step the same values have to be computed:

- The input of the encryption/decryption block (core_in)
- The value send to the next step (to_next)
- The output of that mode-step (mode_out)

which depend on the following variables:

- The input of that mode-step (mode_in)
- The value received from the previous step (from_previous)
- The output of the encryption/decryption block (core_out)

The above process is depicted in Figure 36.

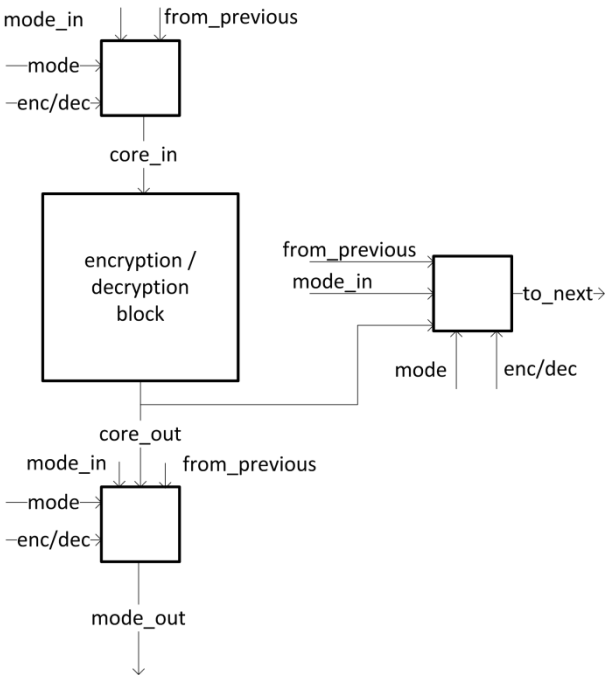


Figure 36: *Operation Mode Procedure*

So three functions are generated (core_in, to_next, mode_out) having as inputs the running mode and the aforementioned variables (mode_in, from_previous, core_out).

3.3.6 Block Diagrams

In this section the block diagrams of the DES&AES engine are presented. The block diagram of the entire engine is depicted in Figure 37. In this block diagram the optional key generators for each algorithm as well as the module implementing the operation modes which contains the two cryptographic engines are illustrated. Furthermore, the ready generator module is depicted which counts the operation cycles and generates certain control signals.

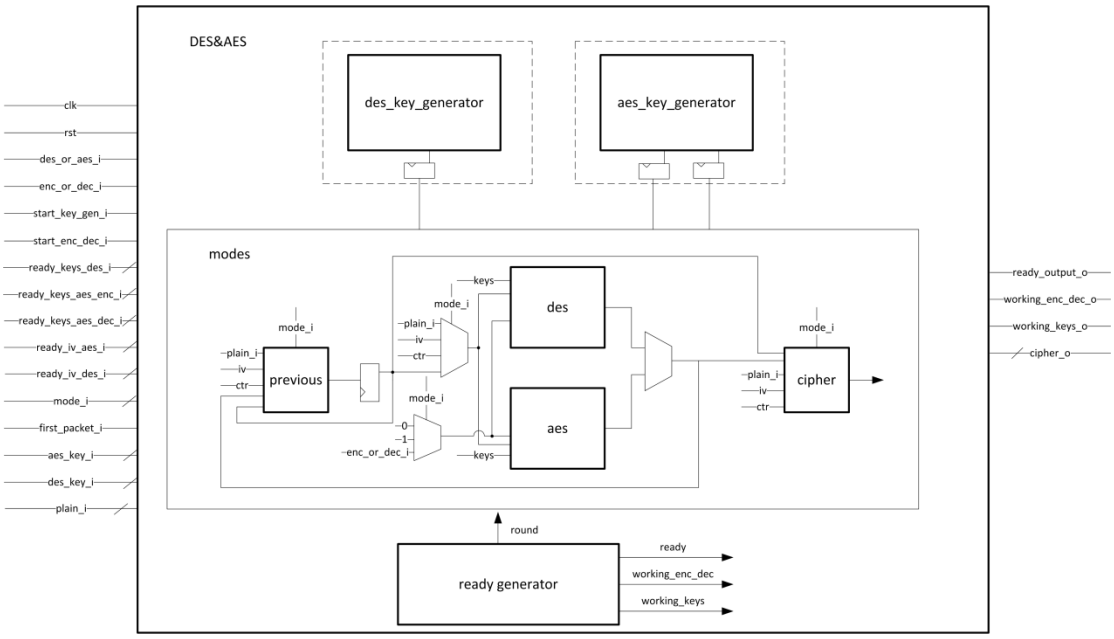


Figure 37: *DES&AES Engine Block Diagram*

The block diagram of the DES SO module is depicted in Figure 38. In this block diagram the two permutation modules, as well as the sixteen instances of the encryption round module are illustrated.

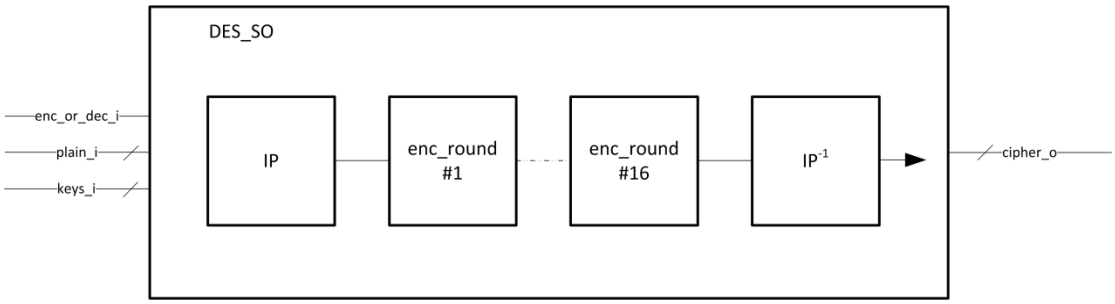


Figure 38: *DES SO Block Diagram*

The block diagram of the DES AO module is depicted in Figure 39. In this block diagram the two permutation modules, as well as the encryption round module are illustrated. In this case there is only one instance of the encryption round module reused in each round (16 rounds in total).

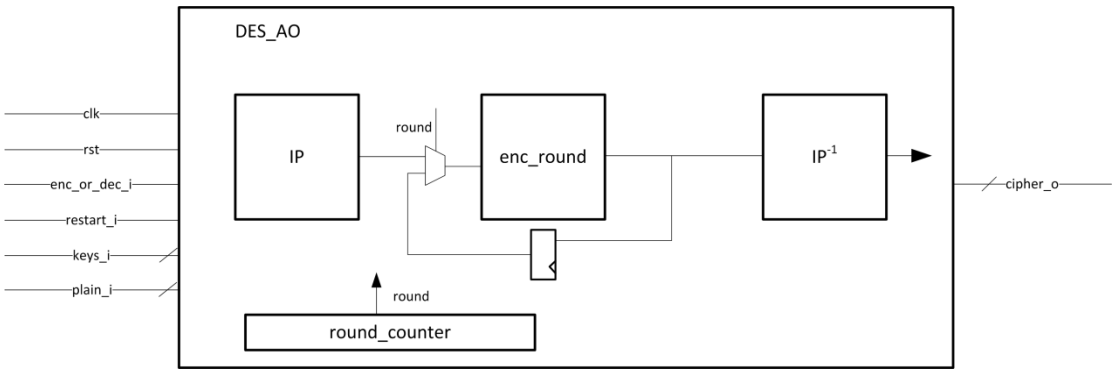


Figure 39: *DES AO Block Diagram*

The block diagram of the AES SO module is depicted in Figure 40. In this block diagram the initial round and final round modules, as well as the m (m depends on the size of the key) instances of the encryption round module are illustrated.

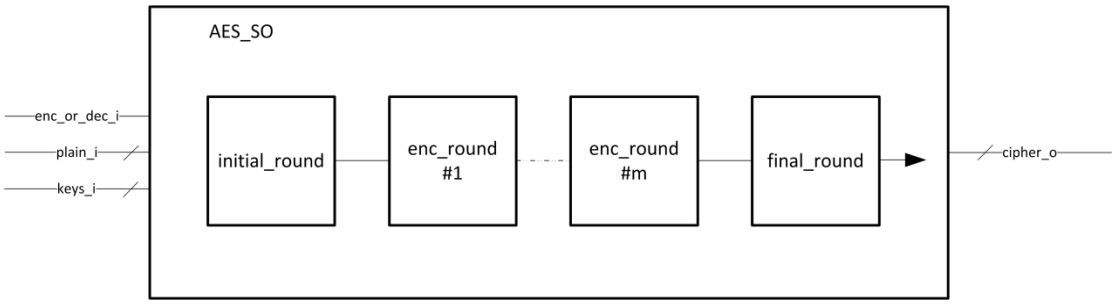


Figure 40: *AES SO Block Diagram*

The block diagram of the AES AO module is depicted in Figure 41. In this block diagram the initial round and final round modules, as well as the encryption round module are illustrated. In this case there is only one instance of the encryption round module reused in each round (m rounds in total).

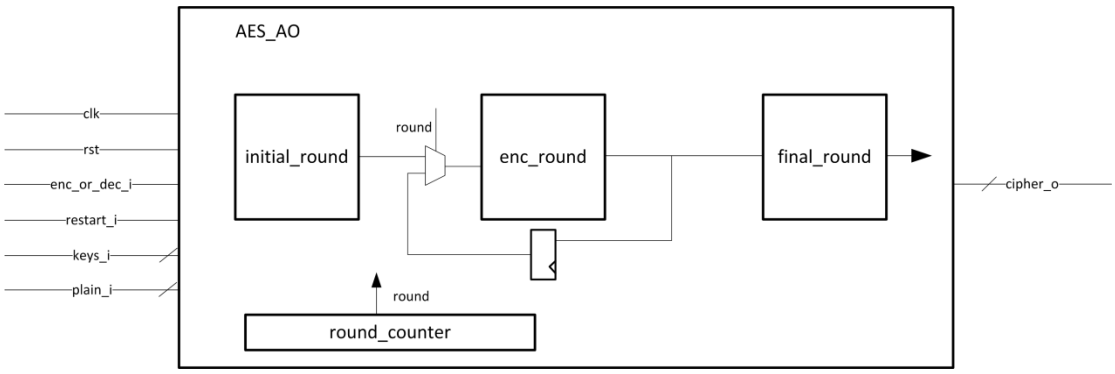


Figure 41: *AES AO Block Diagram*

3.4 Implementation Results

After implementing and synthesizing both DES and AES versions in various ASIC technologies in order to measure its performance, its main characteristics are presented. The most important parameters which have to be mentioned are the area and the frequency of the module, which are results of the synthesis, as well as the cycles required for each core to operate.

The parameters used in the synthesizer tool of Synopsys are the following:

- compile ultra
- no auto ungroup
- timing high effort script
- typical operating conditions

The implementation results of all the main cores which can be used in the DES&AES engine are summarized in Table 15 and Table 16 and illustrated in Figure 42, Figure 43, Figure 44 and Figure 45.

Table 15: *Results of DES Main Cores Implementation*

Core	Technology	Area (mm^2)	Frequency (MHz)	Cycles	Total Time (ns)
DES AO	tsmc (90nm)	0.024	1000.00	16	16.00
DES SO	tsmc (90nm)	0.124	166.67	1	6.00
DES SO	faraday (65nm)	0.087k	238.10	1	4.20

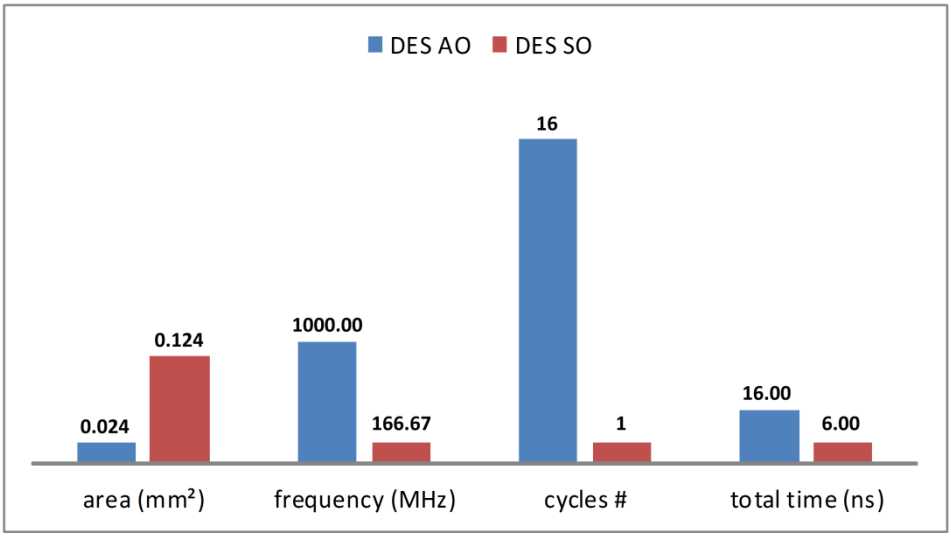


Figure 42: *Results of DES Main Cores Implementation (tsmc 90nm)*

As expected the speed optimized version operates in less total time and the area optimized occupies significantly less area. The fact that the area optimized version reuses only one instance of the encryption round module explains the higher frequency and the more operation cycles.

Table 16: *Results of AES Main Cores Implementation*

Core	Technology	Area (mm^2)	Frequency (MHz)	Cycles	Total Time (ns)
AES AO | 128	tsmc (90nm)	0.393	476.19	9	18.90
AES SO | 128	tsmc (90nm)	1.907	71.43	1	14.00
AES SO | 128	faraday (65nm)	0.991	125.00	1	8.00
AES AO | 192	tsmc (90nm)	0.434	476.19	11	23.10
AES SO | 192	tsmc (90nm)	2.339	58.82	1	17.00
AES SO | 192	faraday (65nm)	1.212	100.00	1	10.00
AES AO | 256	tsmc (90nm)	0.399	476.19	13	27.30
AES SO | 256	tsmc (90nm)	2.604	47.62	1	21.00
AES SO | 256	faraday (65nm)	1.409	83.33	1	12.00

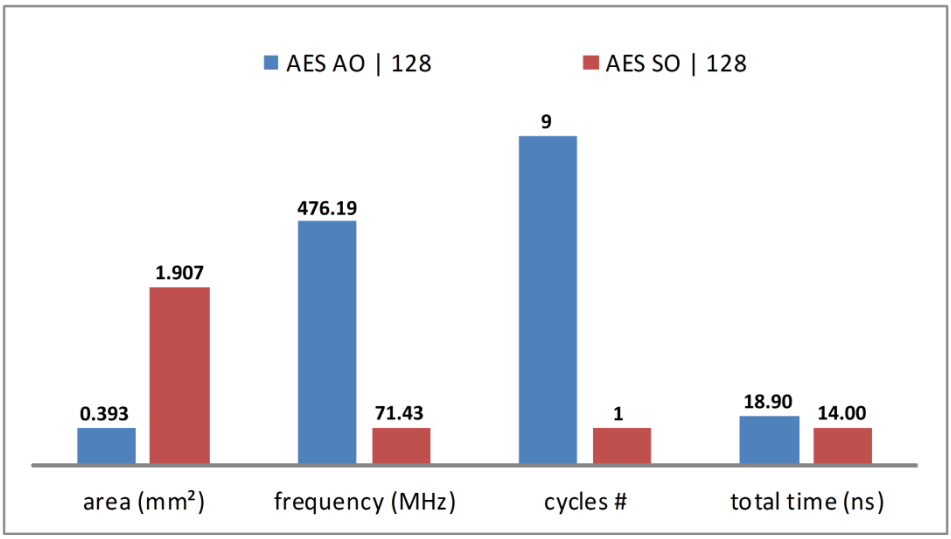


Figure 43: *Results of AES (128-bit key) Main Cores Implementation (tsmc 90nm)*

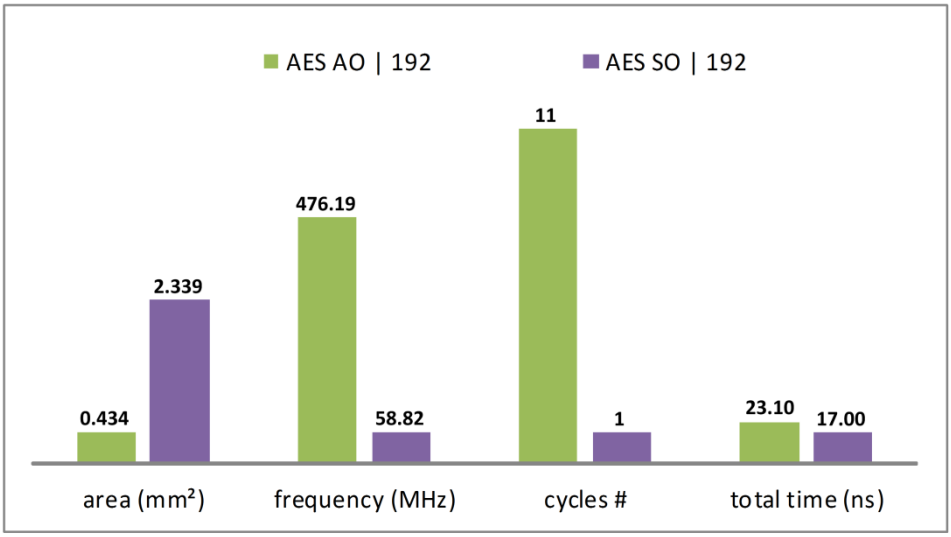


Figure 44: *Results of AES (192-bit key) Main Cores Implementation (tsmc 90nm)*

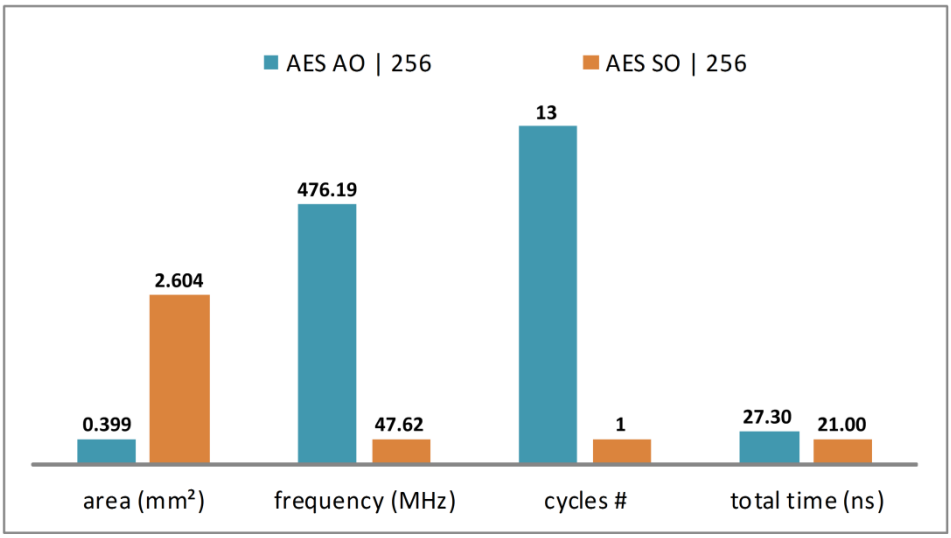


Figure 45: *Results of AES (256-bit key) Main Cores Implementation (tsmc 90nm)*

As mentioned above the speed optimized versions operate in less total time and the area optimized occupy significantly less area. The fact that the area optimized versions reuse only one instance of the encryption round module explains the higher frequency and the more operation cycles.

Finally the implementation results of the key generator cores which can be used in the DES&AES engine are summarized in Table 17.

Table 17: *Results of DES and AES Key Generator Cores Implementation*

Core	Technology	Area (mm^2)	Frequency (MHz)	Cycles	Total Time (ns)
DES Key Generator	tsmc (90nm)	0.014	1000.00	1	1.00
AES Key Generator | 128	tsmc (90nm)	0.114	476.19	10	21.00
AES Key Generator | 192	tsmc (90nm)	0.171	476.19	8	16.80
AES Key Generator | 256	tsmc (90nm)	0.171	476.19	13	27.30

In order to synthesize the key generator modules, the frequency was set to the highest value used in the main cores and it is not the highest one possible. The reason for this is that the key generator module is rarely used and its only requirement is to operate in the same frequency as the main core occupying the less possible area.

4 IDEA Engine

4.1 Introduction

In this chapter, one of the hardware accelerators used in this IP is presented, which implements the IDEA algorithm (mentioned in section 1.3.3). In section 4.2 the compile time parameters of the engine which can be configured by the user to modify the engine according to the specifications and requirements are displayed. In section 4.3 the details of the algorithm's implementation are given. The main parameters considered during the implementation were the area and the frequency of the engine. In section 4.4 the implementation's results in ASIC technologies are illustrated.

4.2 Configuration Parameters

There are two compile time configuration parameters in the IDEA engine which are explained below:

- idea_version: this parameter defines which version of the module will be used, the speed optimized or the area optimized.
- use_key_generator: this parameters defines whether the key generator module will be used or not.

4.3 Implementation

4.3.1 General Description

There are two versions implementing the IDEA algorithm, the speed optimized (SO) and the area optimized (AO). The basic features of the IDEA engine are the following:

- Processes 64-bit data blocks with 128-bit key.
- Fully synchronous design
- Encryption and decryption unit in single core
- Two versions are available for the user to select:
    - Area Optimized version (**small area/resources** utilization)
    - Speed Optimized version (fully **pipelined**)
- Key generator modules included
- Key generator can be ignored using ready keys inserted from the user
- External memory not required
- Available signals to indicate when input data can be inserted an when the output is ready

4.3.2 Pin Description

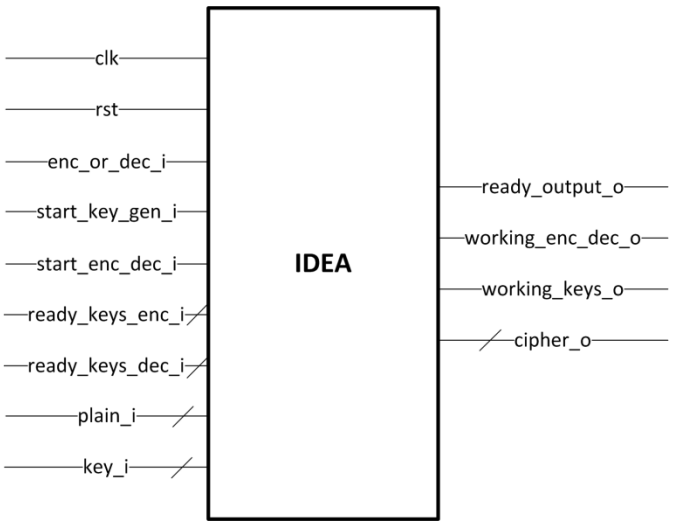


Figure 46: *IDEA Engine Symbol Diagram*

Table 18 contains the description of each input/output pin existing in the IDEA engine.

Table 18: *IDEA Engine Pin Description*

Name	Type	Width (bits)	Description
clk	input	-	clock signal
rst	input	1	reset signal (resets when LOW)
enc_or_dec_i	input	1	LOW for encryption / HIGH for decryption
start_key_gen_i	input	1	HIGH to start a new key generation
start_enc_dec_i	input	1	HIGH to start a new encryption / decryption
ready_keys_enc_i	input	832	ready encryption keys if key generator not used
ready_keys_dec_i	input	832	ready decryption keys if key generator not used
plain_i	input	64	input data
key_i	input	128	encryption key
cipher_o	output	64	output data
ready_output_o	output	1	HIGH when output is ready
working_enc_dec_o	output	1	LOW when encryption / decryption finished
working_keys_o	output	1	LOW when key generation finished

4.3.3 Process of Key Generation and Encryption / Decryption

The following sequence of steps has to be executed to perform a key generation:

1) Wait until *working_enc_dec_o* and *working_keys_o* are LOW
2) Set *key_i*
3) Set *start_key_gen_i* (pulse) to start the key generation process

The following sequence of steps has to be executed to perform an encryption/decryption:

1) Wait until *working_enc_dec_o* and *working_keys_o* are LOW
2) Set *enc_or_dec_i* to select between encryption and decryption, *plain_i*
3) Set *start_enc_dec_i* (pulse) to start the encryption/decryption process
4) Wait until *ready_output_o* is HIGH to read the result of the process

Note that no step can be executed before the previous one is finished. Also, if the key generator is not used, there are no signals *key_i*, *working_keys_o*, and the key update is performed by setting the signals *ready_keys_enc_i, ready_keys_dec_i* and then setting the signal *start_key_gen_i* (pulse) when *working_enc_dec_o* is LOW.

4.3.4 Algorithmic Details

As mentioned in section 1.3.3.1 two of the main algebraic operations used in IDEA algorithm are the following:

- Addition of integers modulo (2^{16}) with inputs and outputs treated as unsigned 16-bit integers
- Multiplication of integers modulo ($2^{16}+1$) with inputs and outputs treated as unsigned 16-bit integers

The addition modulo 2^{16} can be easily implemented as it is equal to the normal addition ignoring the output carry. In contrast, the multiplication modulo $2^{16}+1$ is very hard to be implemented. There are many solutions to this problem and one of the most efficient is the dedicated modulo multiplier implemented in the Microprocessors and Digital Systems Lab of NTUA called “fast_16bits_mult_mod”.

Another implementation difficulty can be found in the production of the decryption keys. As mentioned in Table 2: *Decryption Subkeys Generation Table* in order to produce the decryption keys, one of the following operations is applied in the encryption keys:

- multiplicative inverse modulo $2^{16}+1$ (encryption_key $^{-1}$ modulus $2^{16}+1$)
- additive inverse modulo 2^{16} (-encryption_key modulus 2^{16})

The additive inverse modulo 2^{16} can be easily implemented by just applying the operator "-" in the encryption key and keep the last 16 bits of the result. But in order to implement the highly demanding multiplicative inverse modulo $2^{16}+1$ an efficient method had to be used.

The design of inverse modulo ($2^{16}+1$) multiplier is done using a novel realization of the power algorithm for Euler's theorem, which results in the fast inverse modulo multiplier. Euler's Totient function is written as $\phi(m)$.

According to Euler's theorem, if *a* is coprime to *m*, that is, gcd(*a*, *m*) = 1, then

- $a^{\phi(m)} \equiv 1 \pmod m$

This follows from the fact that *a* belongs to the multiplicative group (Z/mZ)* if and only if *a* is coprime to *m*. Therefore the modular multiplicative inverse can be found directly:

- $a^{\phi(m)-1} \equiv a^{-1} \pmod m$

In the special case when *m* is a prime then:

- $\phi(m) = m-1$.

So, the modular inverse is given by the above equation as:

- $a^{-1} \equiv a^{m-2} \pmod m$

So in order to produce the decryption keys which require the multiplicative inverse operation, the above algorithm can be used because $2^{16}+1$ is a prime number and as a result:

- decryption_key = encryption_key $^{65535} \pmod{2^{16}+1}$

In order to compute the above relation efficiently the algorithm Square and Multiply, described in Table 19 is used.

Table 19: *Square and Multiply Algorithm*

Operation	$C = A^B \pmod N$
No of steps	k-1

Algorithm	Z:= A;
	
	if (B[0]=0) then
	  C:= 1;
	else
	  C:= A;
	end
	
	for (i:= 1 to k-1) do
	  Z:= $Z^2 \pmod N$;
	  if (B[i]=1) then
	  C:= $C \cdot Z \pmod N$;
	  end
	end
	
	return C;

However in this particular case, variable B is a known number (2^{16}-1) which has all its bits set to 1. Also the multiplications modulo 2^{16}+1 in the block of the for-loop are computed using the aforementioned modulo multiplier "fast_16bits_mult_mod". So the algorithm is modified as shown in Table 20.

Table 20: *Modified Square and Multiply*

Operation	$C = A^B \pmod N$
No of steps	15

Algorithm	Z:= A;
	C:= A;
	
	for (i:= 1 to 15) do
	  Z:= fast_16bits_mult_mod(Z,Z);
	  C:= fast_16bits_mult_mod(C,Z);
	end
	return C;

4.3.5 Implementation Details

There are two different implementations of the IDEA algorithm, as mentioned in section 4.3.1, the speed optimized (SO) and the area optimized (AO). The first one (SO) generates eight instances of the main encryption round and one instance of the output transformation round. These sub-modules are wired and operate in one cycle. The second version (AO) reuses a single cell which implements the main encryption round. To implement the output transformation round there are multiplexers that ignore steps 5 to 14 when the current round is the last one.

4.3.6 Block Diagrams

In this section the block diagrams of the IDEA engine are presented. The block diagrams of the two versions (speed optimized and area optimized) are depicted in Figure 47 and Figure 48 respectively.

In both figures the optional key generator, as well as the main module implementing the encryption/decryption operation are illustrated. In Figure 47 the main module contains eight instances of the encryption round module and one instance of the output transformation round. In Figure 48 the main module contains one instance of the encryption round module which is reused in each round (9 rounds in total).

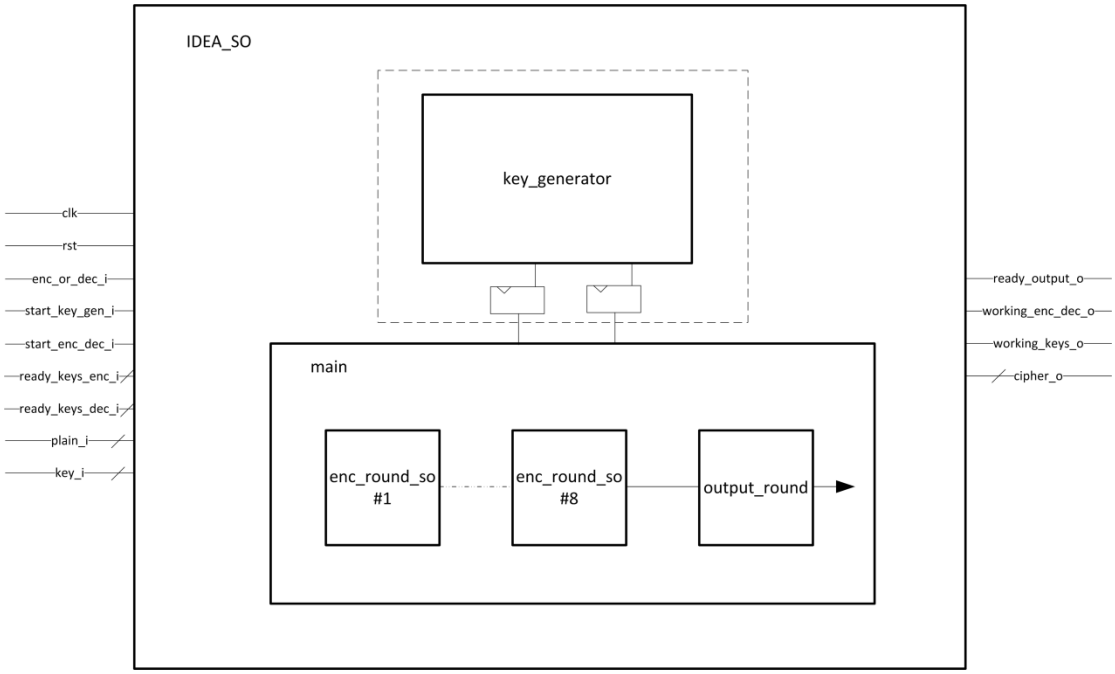


Figure 47: *IDEA SO Block Diagram*

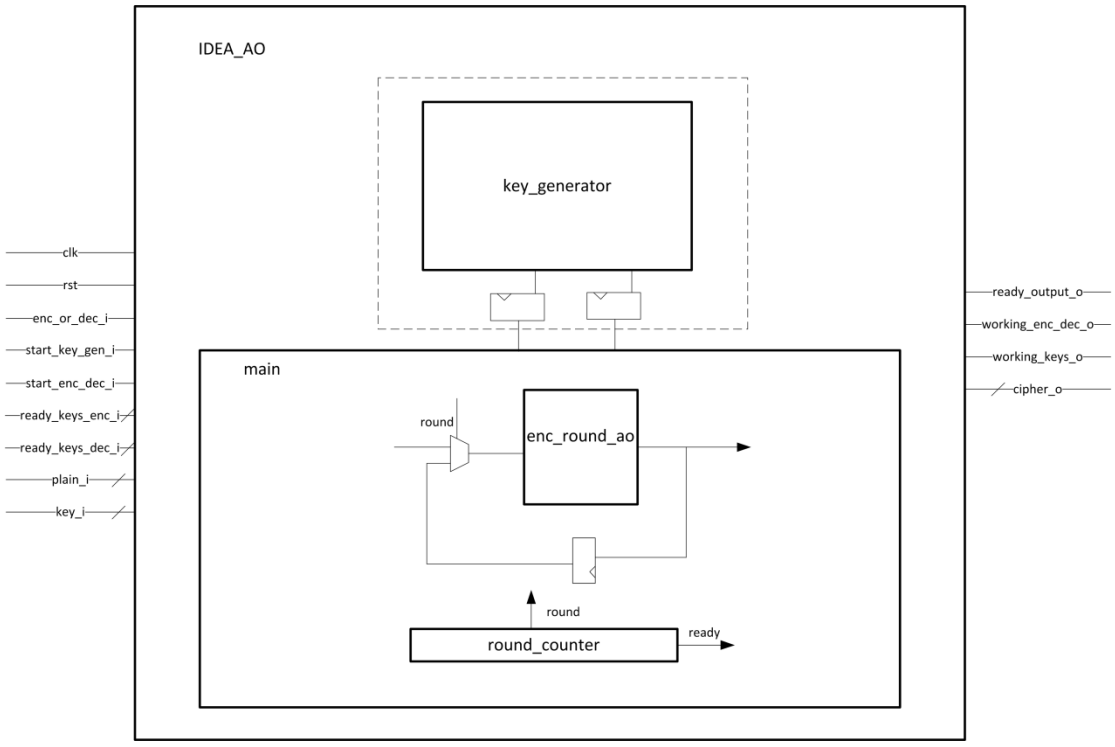


Figure 48: *IDEA AO Block Diagram*

4.4 Implementation Results

After implementing and synthesizing both IDEA versions in various ASIC technologies in order to measure its performance, its main characteristics are presented. The most important parameters which have to be mentioned are the area and the frequency of the module, which are results of the synthesis, as well as the cycles required for each core to operate.

The parameters used in the synthesizer tool of Synopsys are the following:

- compile ultra
- no auto ungroup
- timing high effort script
- typical operating conditions

The implementation results of all the main cores which can be used in the IDEA engine are summarized in Table 21 and illustrated in Figure 49.

Table 21: *Results of IDEA Main Cores Implementation*

Core	Technology	Area (mm^2)	Frequency (MHz)	Cycles	Total Time (ns)
IDEA AO	tsmc (90nm)	0.053	163.93	9	54.90
IDEA SO	tsmc (90nm)	0.286	23.26	1	42.99
IDEA AO	faraday (65nm)	0.037	357.14	9	25.20
IDEA SO	faraday (65nm)	0.198	49.75	1	20.10

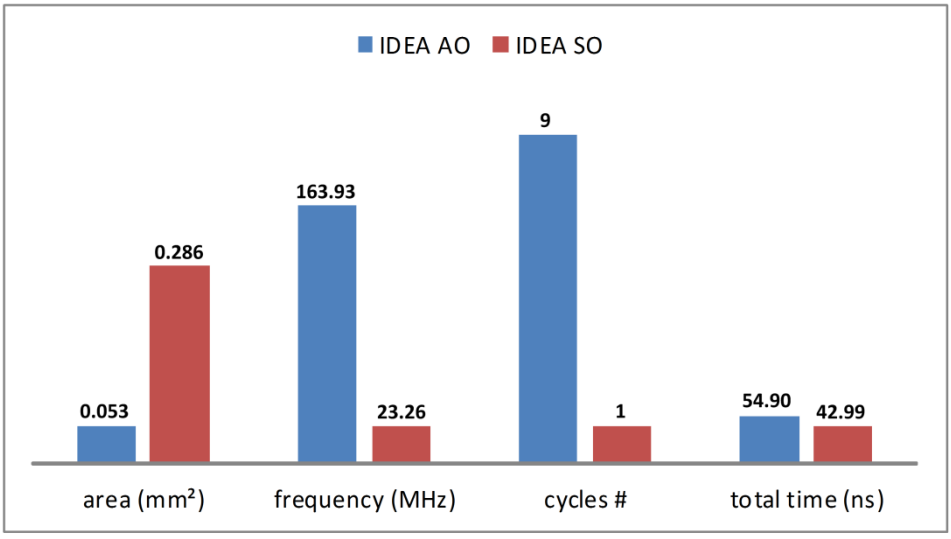


Figure 49: *Results of IDEA Main Cores Implementation (tsmc 90nm)*

As expected the speed optimized version operates in less total time and the area optimized occupies significantly less area. The fact that the area optimized version reuses only one instance of the encryption round module explains the higher frequency and the more operation cycles.

The implementation results of the key generator core which can be used in the IDEA engine are summarized in Table 22.

Table 22: *Results of IDEA Key Generator Core Implementation*

Core	Technology	Area (mm^2)	Frequency (MHz)	Cycles	Total Time (ns)
IDEA Key Generator	tsmc (90nm)	0.193	163.93	15	91.50
IDEA Key Generator	faraday (65nm)	0.091	357.14	15	42.00

In order to synthesize the key generator module, the frequency was set to the highest value used in the main cores and it is not the highest one possible. The reason for this is that the key generator module is rarely used and its only requirement is to operate in the same frequency as the main core occupying the less possible area.

5 RSA Engine

5.1 Introduction

In this chapter, one of the hardware accelerators used in this IP is presented, which implements the RSA algorithm (mentioned in section 1.4.1). In section 5.2 the compile time parameters of the engine which can be configured by the user to modify the engine according to the specifications and requirements are displayed. In section 5.3 the details of the algorithm's implementation are given. The main parameters considered during the implementation were the area and the frequency of the engine. In section 5.4 the implementation's results in ASIC technologies are illustrated.

5.2 Configuration Parameters

There are three compile time configuration parameters in the RSA engine which are explained below:

- rsa_bit_size: this parameter defines the size of the RSA key and as a result some other variables which are the plaintext and the ciphertext. The valid values of this parameter are: 512/1024/2048/4096
- mult_unit_size: this parameter modifies the operation cycles of the algorithm and as a result its area. The higher the value of this parameter, the lower the operation cycles of the engine. However, as the cycles are reduced, the area is increased. The valid values of this parameter are all the powers of 2 less or equal to the rsa_bit_size.
- use_key_generator: this parameters defines whether the key generator module will be used or not.

5.3 Implementation

5.3.1 General Description

The basic features of the RSA engine are the following:

- Parameterized operation cycles / area
- Key generator module included (for the private key)
- Key generator can be ignored using ready private key inserted from the user
- External memory not required
- Available signals to indicate when input data can be inserted an when the output is ready
- Key size supported : 512 / 1024 / 2048 / 4096
- Encryption and decryption unit in single core
- Fully synchronous design

5.3.2 Pin Description

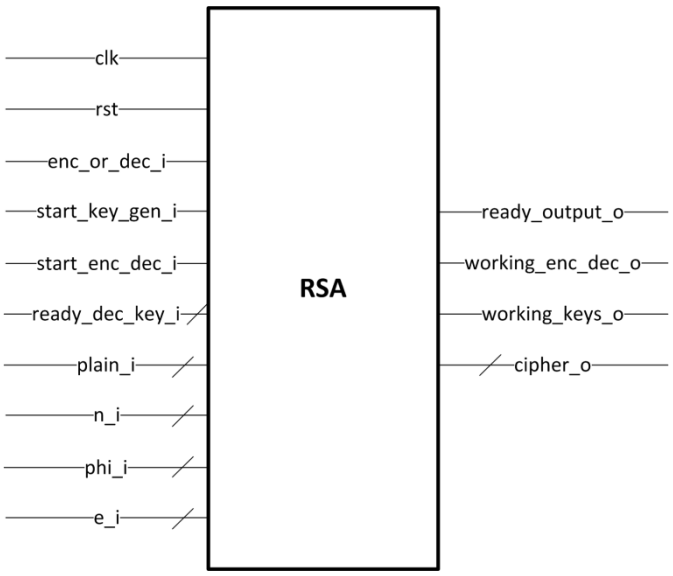


Figure 50: *RSA Engine Symbol Diagram*

Table 23 contains the description of each input/output pin existing in the RSA engine.

Table 23: *RSA Engine Pin Description*

Name	Type	Width (bits)	Description
clk	input	-	clock signal
rst	input	1	reset signal (resets when LOW)
enc_or_dec_i	input	1	LOW for encryption / HIGH for decryption
start_key_gen_i	input	1	HIGH to start a new key generation
start_enc_dec_i	input	1	HIGH to start a new encryption / decryption
plain_i	input	512/1024/2048/4096	input data
n_i	input	512/1024/2048/4096	product p · q
phi_i	input	512/1024/2048/4096	product (p-1) · (q-1)
e_i	input	512/1024/2048/4096	public key exponent
ready_dec_key_i	input	512/1024/2048/4096	ready private key exponent if key generator not used
cipher_o	output	512/1024/2048/4096	output data
ready_output_o	output	1	HIGH when output is ready

working_enc_dec_o	output	1	LOW when encryption / decryption finished
working_keys_o	output	1	LOW when key generation finished

5.3.3 Process of Key Generation and Encryption / Decryption

The following sequence of steps has to be executed to perform a key generation:

1) Wait until *working_enc_dec_o* and *working_keys_o* are LOW
2) Set *e_i* , *phi_i*, *n_i* (no specific order required)
3) Set *start_key_gen_i* (pulse) to start the key generation process

The following sequence of steps has to be executed to perform an encryption/decryption:

1) Wait until *working_enc_dec_o* and *working_keys_o* are LOW
2) Set *enc_or_dec_i* to select between encryption and decryption, *plain_i*, *n_i* (only for encryption), *e_i* (only for encryption)
3) Set *start_enc_dec_i* (pulse) to start the encryption/decryption process
4) Wait until *ready_output_o* is HIGH to read the result of the process

Note that no step can be executed before the previous one is finished. Also, if the key generator is not used, there are no signals *phi_i*, *working_keys_o*, and the key update is performed by setting the signal *ready_dec_key_i* and then setting the signal *start_key_gen_i* (pulse) when *working_enc_dec_o* is LOW.

5.3.4 Algorithmic Details

In the RSA algorithm there are some parts that their implementation is worth to be mentioned. The main part of the RSA encryption/decryption is the computation of the following:

- $C = M^E \pmod N$ (encryption)
- $M = C^D \pmod N$ (decryption)

So it is important to find an efficient way to implement this computation. One of the most known and effective algorithms for this, is the one called “Square and Multiply” which is described in Table 19: *Square and Multiply Algorithm*.

However, the implementation of this algorithm has its own difficulties as it requires the computation of the following:

- $C = C \cdot Z \pmod N$

So an effective algorithm for the above computation has to be used. The algorithm the most industries use for this purpose is the one called “Montgomery Multiplication” which is described in Table 24.

Table 24: *Montgomery Multiplication Algorithm*

Operation	$R = A \cdot B \cdot 2^{-k} \pmod N$ (k is the rsa_bit_size)
No of steps	k+1

Algorithm	R:= 0;
	Q:= 0;
	
	for (i:= 0 to k) do
	  Q:= R[0];
	  R:= (R + Q·N + A[i]·B) div 2;
	end
	
	if (R ≥ N) then
	  R:= R−N;
	end
	
	return R;

As it is obvious the output of this algorithm is not the one needed for the “Square and Multiply” because of the presence of the factor 2^{-k} in it.

To solve this problem the initial values in the “Square and Multiply” algorithm have to be transformed in the Montgomery field. For example, the number a transformed in the Montgomery field is the number $A_m = A \cdot 2^k \pmod N$. The transformation of any number in the field of natural number to the Montgomery field can be performed by executing the Montgomery multiplication of $2^{2k} \pmod N$ and this number:

- Montgomery $(2^{2k} \pmod N, A) = (A \cdot 2^{2k} \pmod N \cdot 2^{-k}) \pmod N = A \cdot 2^k \pmod N$

The reason why this is the solution of the problem, is because an operation with operands in the Montgomery field has an output in the same field as shown below.

- $A_m = A \cdot 2^k \pmod N$ (A_m is the transformation of a in the Montgomery field)
- $B_m = B \cdot 2^k \pmod N$ (B_m is the transformation of b in the Montgomery field)
- C_m = Montgomery (A_m, B_m) = $A_m \cdot B_m \cdot 2^{-k} \pmod N = A \cdot 2^k \cdot B \cdot 2^k \cdot 2^{-k} \pmod N = A \cdot B \cdot 2^k \pmod N = (A \cdot B)_m$

So, in the "Square and Multiply" algorithm, instead of initializing Z and C to the value M, they are initialized to $M \cdot 2^k \pmod N$. The initialization of C to the value 1 never occurs in the RSA so C is always initialized to $M \cdot 2^k \pmod N$.

By doing this, the output C of the "Square and Multiply" is in the Montgomery field so at the end of the algorithm the output has to be transformed in the field of natural numbers. This can be done by executing a Montgomery multiplication of the output C with the number 1:

- Montgomery (C_m,1) = $C_m \cdot 1 \cdot 2^{-k} \pmod N = C \cdot 2^k \cdot 2^{-k} \pmod N = C \pmod N$

As mentioned, the transformation of any natural number to the Montgomery field requires the computation of the value: $2^{2k} \pmod N$. To perform this operation efficiently a 2-stage algorithm can be executed.

In the first stage the value t is computed, where t satisfies the following relations:

- $2^t < N$ and
- $2^{t+1} \geq N$

The algorithm for the above process is described in Table 25.

Table 25: *1st Stage of $2^{2k} \pmod N$ Computation*

Operation	max t: $2^t < N$
No of steps	≤ k

Algorithm	t:= k;
	
	while (N[k] != 0) do
	  t:= k-1;
	end
	
	return t;

In the second stage the value 2^{2k} (mod N) is computed by doubling in each step the initial value (2^t) and if the result is equal or greater than N, it is reduced by N. The algorithm for the above process is described in Table 26.

Table 26: *2nd Stage of 2^{2k} (mod N) Computation*

Operation	t_2_2k = 2^{2k}(mod N)
No of steps	2k-t

Algorithm	t_2_2n:= 2^t; //t is the output from stage 1
	
	for (i:= t to 2k-1) do
	t_2_2n = t_2_2n · 2;
	if (t_2_2n ≥ N) then
	t_2_2n = t_2_2n - N;
	end
	end
	
	return t_2_2n;

Another important part of the RSA algorithm is the computation of the decryption key D:

- $D \equiv E^{-1} \bmod \phi(N)$

Hence, the modified Penk's algorithm [5] is used, because it performs the computation efficiently. Penk's algorithm is described in Table 27.

Table 27: *Penk Algorithm*

Operation	r = a^{-1}(mod p)
No of steps	[k,4k]

Algorithm	u:= p;
	v:= a;
	r:= 0;
	s:= 1;
	
	while (v > 0) do
	if (u is even) then
	if (r is even) then
	u:= u/2;
	r:= r/2;
	else
	u:= u/2;
	r:= (r + p)/2;
	end
	else if (v is even) then
	if (s is even) then
	v:= v/2;
	s:= s/2;
	else
	v:= v/2;

```
                s:= (s + p)/2;
     else
          x:= u - v;
          if (x > 0) then
               u:= x;
               r:= r - s;
               if (r < 0) then
                    r:= r + p;
               end
          else
               v:= -x;
               s:= s - r;
               if (s < 0) then
                    s:= s + p;
               end
          end
     end
end

if (r > p) then
     r:= r - p;
end

if (r < 0) then
     r:= r + p;
end

return r;
```

Penk's algorithm requires the modulus to be odd. At first, this appears to make the operation useless in the case of RSA key generation where the private key exponent D is:

- $D = Penk(E, \phi(N)) = E^{-1} \bmod \phi(N)$

Note that $\phi(N) = (p-1) \cdot (q-1)$ and p and q both prime, so $\phi(N)$ is even.

However, since E must be odd (otherwise no inverse exists), D can be calculated as:

- $D = (1 + (\phi(N) \cdot (E - Penk(\phi(N), E))) / E$

The above relation is very demanding in hardware, so an efficient way of computing this must be found.

The algorithm used to solve the problem is described in Table 28.

Table 28: *(1+ab)/c Algorithm*

Operation	p = (1+a·b)/c, b<c
No of steps	k

Algorithm	//Find min x: $b \cdot 2^x \geq c$
	x:= 0;
	prev_u:= 0;
	prev_p:= 0;
	p:= 0;
	u:= 0;
	d:= b;
	
	while (c ≥ d) do
	  if (a[x] = 1) then
	  u:= u + d;
	  end
	  if (u ≥ c) then
	  u:= u − c;
	  p:= p + 1;
	  end
	  prev_u:= d;
	  d:= d·2;
	  x:= x + 1;
	end
	
	//Now d ≥ c
	for (i:= x to k-1) do
	  temp:= prev_u·2;
	  if (temp ≥ c) then
	  prev_p:= prev_p·2 + 1;
	  prev_u:= temp − d;
	  else
	  prev_p:= prev_p·2;
	  prev_u:= temp;
	  end
	  if (a[i] = 1) then
	  p:= p + prev_p;
	  u:= u + prev_u;
	  end
	  if (u ≥ c) then
	  u:= u − c;
	  p:= p + 1;
	  end
	  d:= d·2;
	end
	
	u:= u + 1;
	
	if (u ≥ c) then
	  p:= p + 1;
	end
	
	return p;

So concerning that:

$$E - Penk(\phi(N), E) < E$$

the above algorithm can be used with:

$a = \phi(N)$

$b = E - Penk(\phi(N), E)$

$c = E$

5.3.5 Implementation Details

The instructions inside the for-loop of the Montgomery algorithm:

```
Q:= R[0];
R:= (R + Q·N + A[i]·B) div 2;
```

realize the main cell of the Montgomery multiplication, called “Montgomery cell”. In order to reduce the operation cycles of the Montgomery multiplication, more than one Montgomery cells can be combined in a new sub-module called “mult_unit”. The number of the combined Montgomery cells in the “mult_unit” sub-module is defined by the parameter *mult_unit_size* mentioned in section 5.2. As a result the operation cycles required for a Montgomery multiplication are divided by *mult_unit_size*.

5.3.6 Block Diagrams

In this section the block diagrams of the RSA engine are presented. The block diagram of the entire engine is depicted in Figure 51. In this block diagram the optional key generator included as well as the module implementing the main operation of RSA are illustrated. In the key generator the inverse_mod module (implementing Penk's algorithm) and the mult_div module (calculating the (1+ab)/c operation) are included. The rsa_main contains the module generating the value 2^{2k}(mod N) and the module implementing the "Square and Multiply" algorithm.

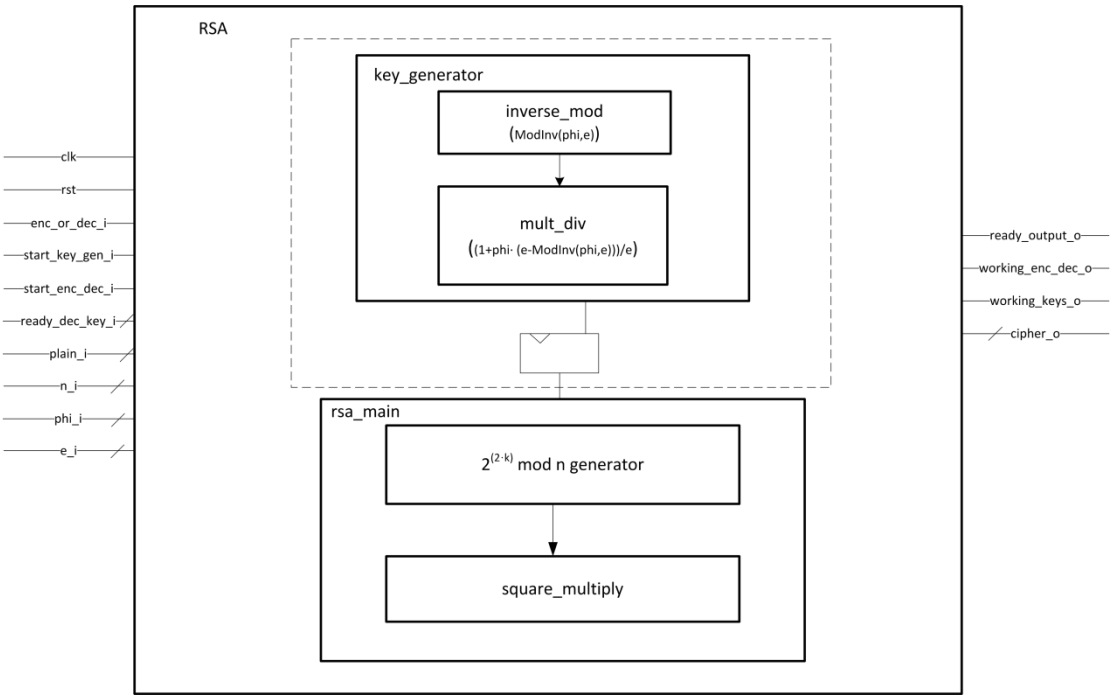


Figure 51: *RSA Engine Block Diagram*

The block diagram of the Square and Multiply sub-module is depicted in Figure 52. In this block diagram there is an instance of the Montgomery algorithm as well as a round counter.

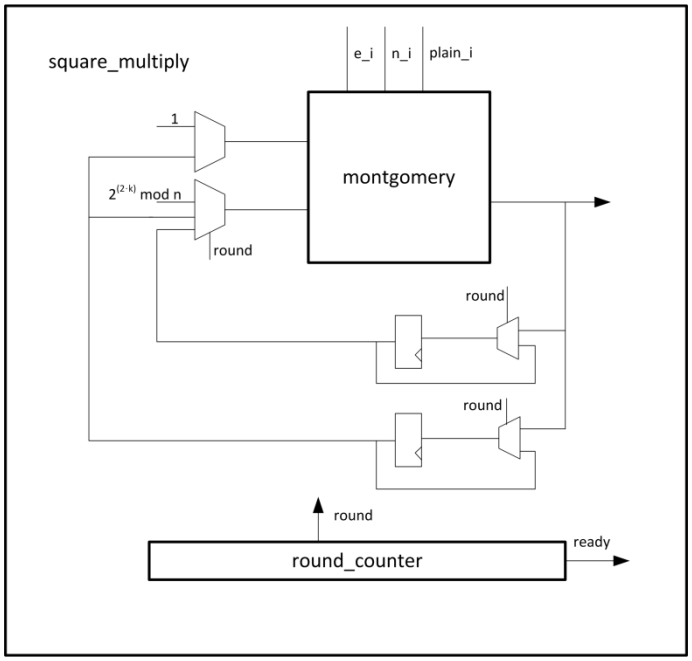


Figure 52: *Square and Multiply Block Diagram*

The block diagram of the Montgomery multiplication sub-module is depicted in Figure 53. In this block diagram the mult_unit module (implementing the main computation of the Montgomery algorithm) as also a round counter are illustrated. In the mult_unit module there are m instances of the Montgomery cell (where m is a configurable parameter as mentioned in section 5.3.5).

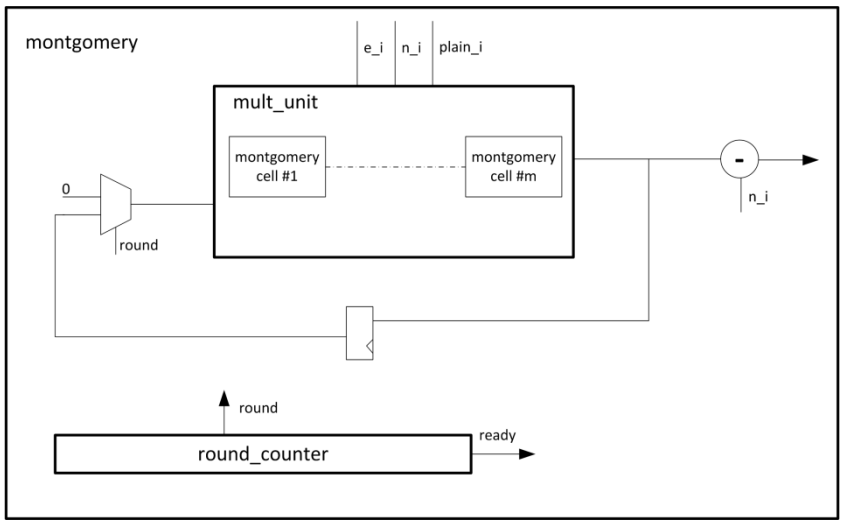


Figure 53: *Montgomery Multiplication Block Diagram*

5.4 Implementation Results

After implementing and synthesizing the RSA engine in various ASIC technologies in order to measure its performance, its main characteristics are presented. The most important parameters which have to be mentioned are the area and the frequency of the module, which are results of the synthesis, as well as the cycles required for each core to operate.

The parameters used in the synthesizer tool of Synopsys are the following:

- compile ultra
- no auto ungroup
- timing high effort script
- typical operating conditions

The implementation results of the main cores which can be used in the RSA engine are summarized in Table 29 and illustrated in Figure 54 and Figure 55.

Table 29: *Results of RSA Main Cores Implementation*

Core	Technology	Area (mm^2)	Frequency (MHz)	Max Cycles	Total Time (ns)
RSA | 1024 | 1	tsmc (90nm)	0.416	322.58	2099206	6507551
RSA | 1024 | 2	tsmc (90nm)	0.461	250.00	1050118	4200472
RSA | 1024 | 8	tsmc (90nm)	0.949	111.11	263302	2369741
RSA | 2048 | 1	tsmc (90nm)	0.819	277.78	8392710	30213514
RSA | 2048 | 2	tsmc (90nm)	0.833	222.22	4197382	18888407
RSA | 2048 | 8	tsmc (90nm)	1.612	83.33	1050886	12611136

Notes:

- In Core field, “RSA | x | y” corresponds to rsa_bit_size x and mult_unit_size y.
- The maximum operation cycles required for the RSA | x | y are computed from the equation below:
    - Cycles = $x+6+(2x+1)\cdot(x/y)$

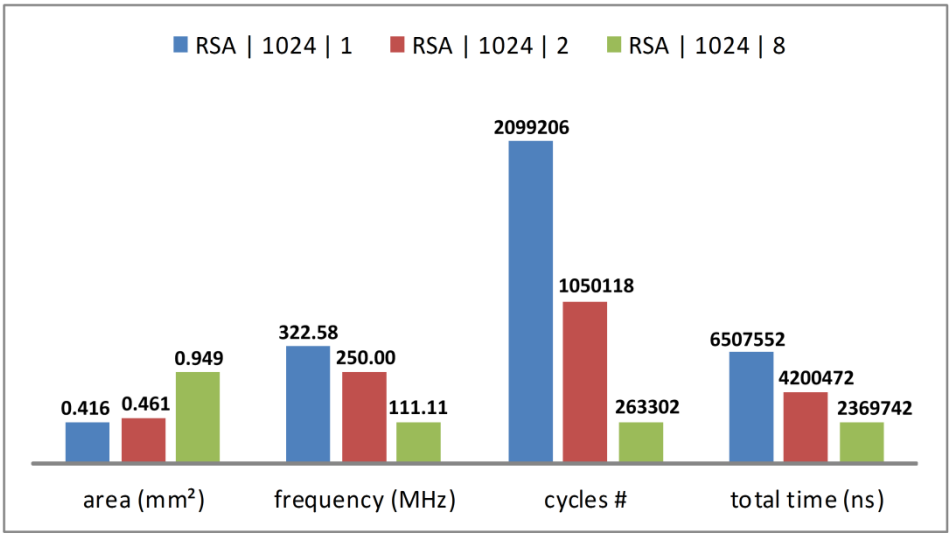


Figure 54: *Results of RSA (1024-bit key) Main Cores Implementation (tsmc 90nm)*

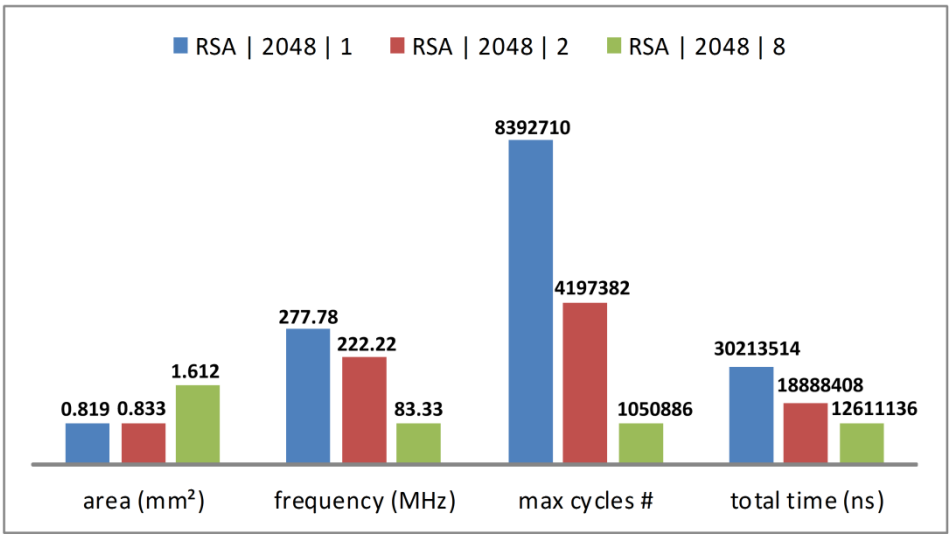


Figure 55: *Results of RSA (2048-bit key) Main Cores Implementation (tsmc 90nm)*

As expected when the parameter *mult_unit_size* is increased the area is also increased and the frequency is decreased, but the operation cycles as well as the total time are significantly decreased.

The implementation results of the key generator cores which can be used in the RSA engine are summarized in Table 30.

Table 30: *Results of RSA Key Generator Cores Implementation*

Core	Technology	Area (mm^2)	Frequency (MHz)	Max Cycles	Total Time (ns)
RSA Key Generator | 1024	tsmc (90nm)	1.046	322.58	3072	9523
RSA Key Generator | 2048	tsmc (90nm)	2.057	277.78	6144	22118

In order to synthesize the key generator modules, the frequency was set to the highest value used in the main cores and it is not the highest one possible. The reason for this is that the key generator module is rarely used and its only requirement is to operate in the same frequency as the main core occupying the less possible area.

Notes:

- In Core field, "RSA Key Generator | x" corresponds to rsa_bit_size x.
- The maximum operation cycles required for the RSA Key Generator |x are computed from the equation below:
    - Cycles = 4x + 3x = 7x (the (1+ab)/c algorithm requires normally x cycles, but in this implementation its cycle is split into three cycles so that the key generator can operate in the frequency of the fastest main core)

6 IP Verification

6.1 Introduction

The three engines described in Chapters 3, 4 and 5 are implemented in Verilog, simulated and tested using Modelsim, and synthesized using Synopsys software tools. However, it is necessary to test the functionality of these engines in real time. The platform used to run these tests is the DE4 Board of Altera with the Stratix IV FPGA (mentioned in section 6.2). In order to control and feed the cryptographic engines an external controller (mentioned in section 6.3) is implemented. This controller is connected with a UART so that the user can monitor the procedure from a terminal.

6.2 FPGA Platform

6.2.1 Key Features

The following hardware is implemented on the DE4 board:

- Featured device
    - Altera Stratix® IV GX FPGA (EP4SGX230C2)
- Configuration status and set-up elements
    - Built-in USB Blaster circuit for programming
    - Fast passive parallel (FPP) configuration via MAX II CPLD and flash memory
    - Three External Programmable PLL timing chip
- Component and interfaces
    - Four Gigabit Ethernet (GigE) with RJ-45 connector
    - Two host and two device Serial ATA (SATA II) ports
    - Two HSMC connectors
    - Two 40-pin expansion headers
    - PCI Express 2.0 (x8 lane) connector

- Memory
    - DDR2 SO-DIMM socket
    - FLASH
    - SSRAM
    - SD Card socket
    - I2C EEPROM
- General user input/output:
    - 8 LEDs
    - 4 push-buttons and 4 slide switches
    - 8-position DIP switch
    - 2 seven-segment displays
- Clock system
    - On-board clock oscillators: 50MHz and 100MHz
    - SMA connectors for external clock input
    - SMA connectors for clock output
- Other interfaces
    - USB 2.0 high-speed host/device OTG
    - Current sensor for FPGA current measurement
    - Temperature sensor

6.2.2 Peripherals

In order to connect the FPGA with a terminal to monitor the process of the demo a UART was implemented. The UART core implements RS-232 asynchronous transmit and receive logic. The UART core sends and receives serial data via the TXD and RXD ports. The I/O buffers on the Altera FPGA do not comply with RS-232 voltage levels, and may be damaged if driven directly by signals from an RS-232 connector. To comply with RS-232 voltage signaling specifications, an external level-shifting buffer is required between the FPGA I/O pins and the external RS-232 connector. Hence, the level shifter MAX232 was used.

6.2.3 Board Overview

In Figure 56 and Figure 57 the top and bottom view of the DE4 board are depicted respectively. The layout of the board is described and the location of the connectors and key components is indicated.

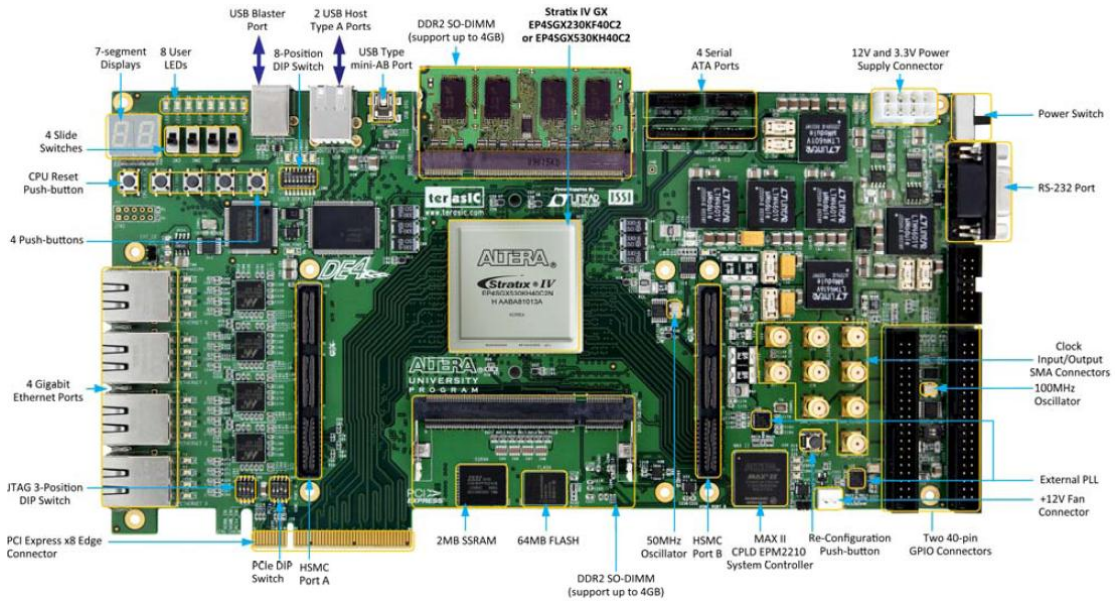


Figure 56: *Top View of the DE4 Board*

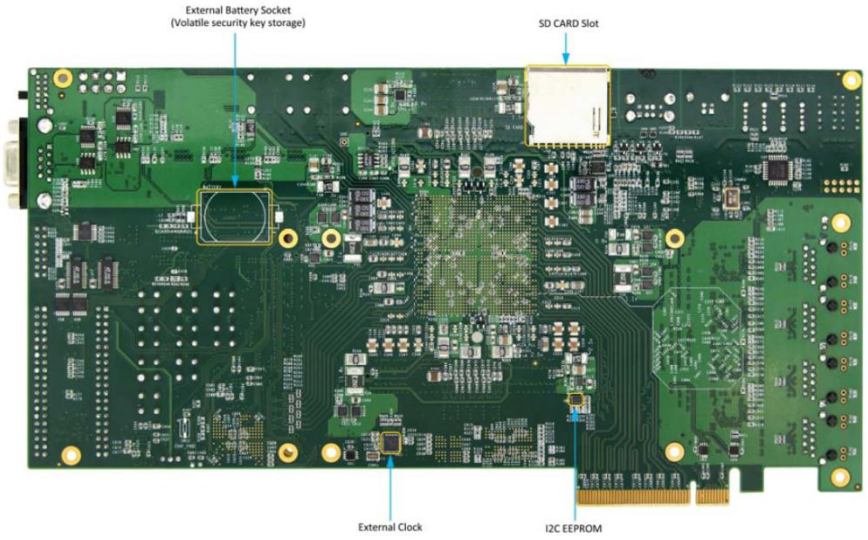


Figure 57: *Bottom View of the DE4 Board*

6.2.4 Block Diagram

The block diagram of the DE4 board is depicted in Figure 58. All key components are connected with the Stratix IV GX FPGA device. Thus, users can configure the FPGA to implement any system design.

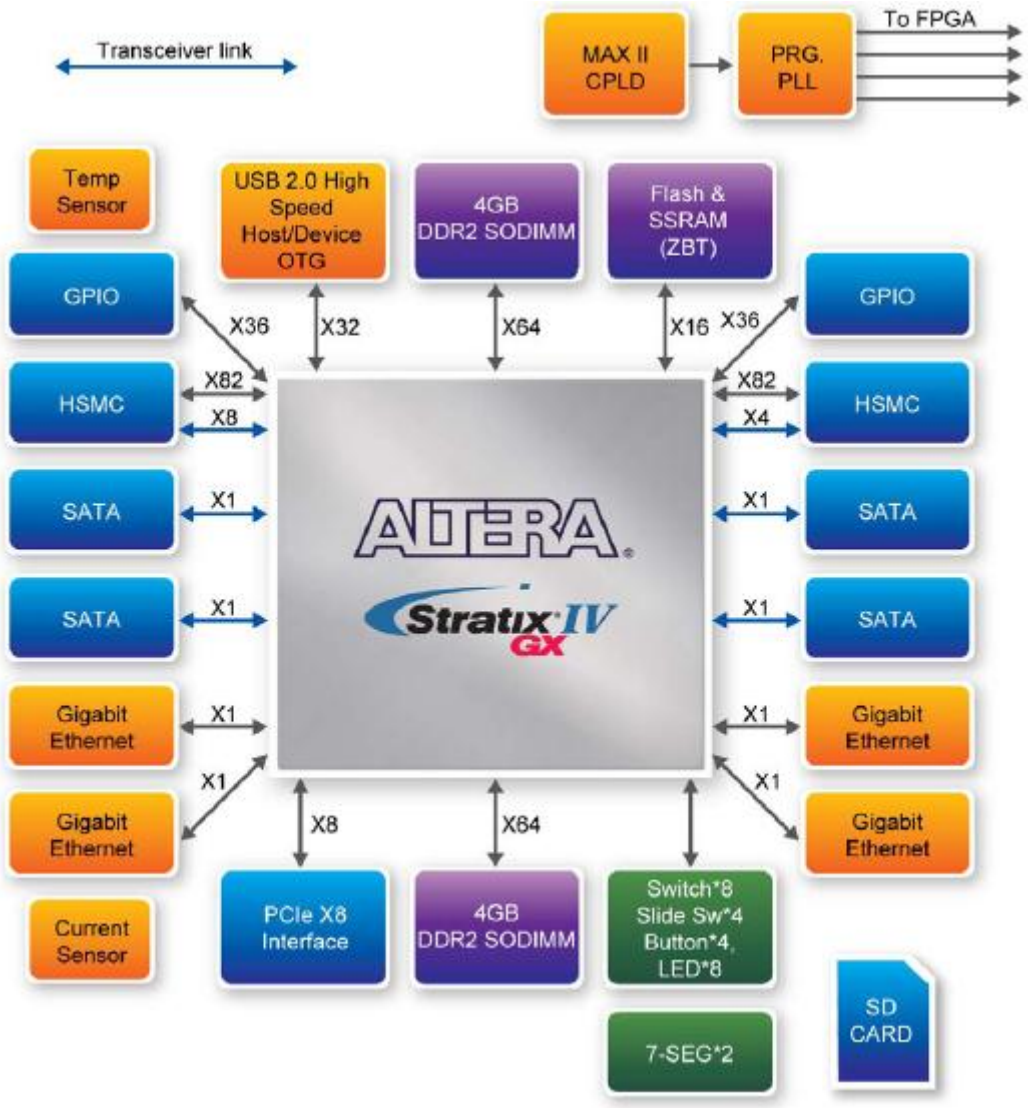


Figure 58: *DE4 Board Block Diagram*

6.3 External Controller

The main operations of the external controller are described below:

- Read the user's input to select the cryptographic algorithm used
- Load the proper tests (stored in the memory of the DE4 board) according to the selected algorithm.
- Feed the selected engine with the proper inputs and collect the results.
- Send monitoring messages in a terminal during the whole procedure using the UART.

The block diagram of the external controller is depicted in Figure 59.

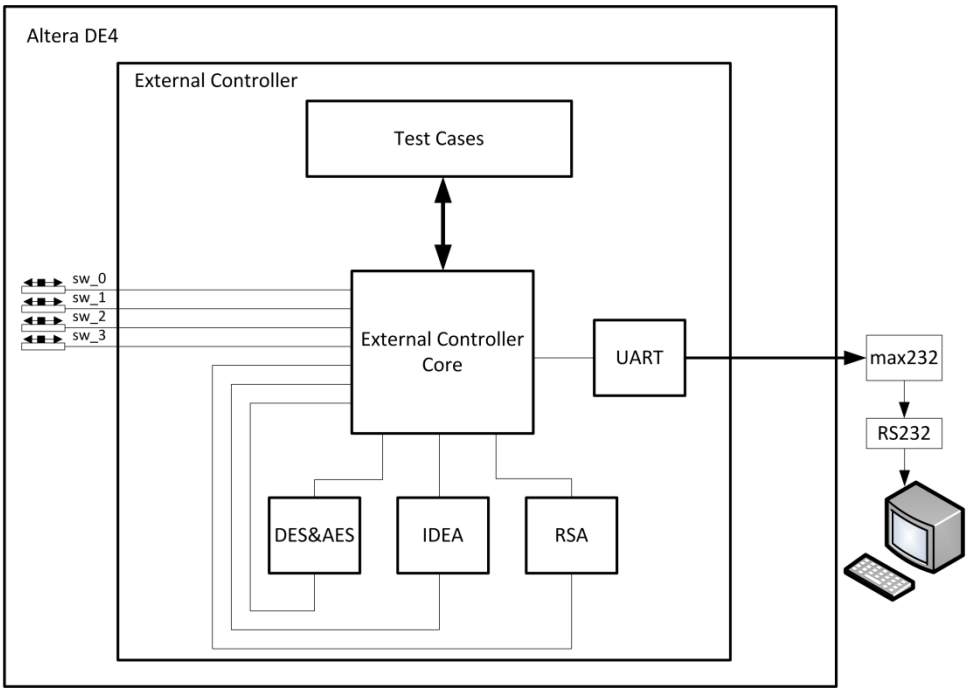


Figure 59: *External Controller Block Diagram*

The use of the External Controller is quite simple. Each slide switch of the DE4 board corresponds to an algorithm as shown below:

- Switch 0: DES
- Switch 1: AES
- Switch 2: IDEA
- Switch 3: RSA

When one of these switches is set to HIGH, the testbench of the corresponding algorithm is executed. If two or more switches are set to HIGH the selected algorithm is the one of the lowest switch.

Conclusion

In this diploma thesis a complete study of certain symmetric-key and public-key cryptographic algorithms was attempted and a presentation of the way these algorithms are integrated in a single cryptographic IP. During the development of this thesis we got familiar with the principals of cryptography and IP architecture while we developed this IP from scratch. We analyzed the basic operations of each algorithm in order to implement them in the most efficient way. In this implementation we had in mind that the circuit should be small, fast and configurable by the user in compile time. The fact that we implemented the entire IP and not only its cryptographic engines gave us the opportunity to learn how a bus interface, a controller and a register file are implemented. Furthermore, the simulation, the synthesis and the testing of the cryptographic engines using a real industrial FPGA board helped us get familiar with a variety of software tools which are highly used in the industry. However the development of this project will continue as the Crypto IP will be integrated into a System on Chip (SoC). For this purpose the drivers for a specific processor used in the SoC will be implemented as well as a DMA.

Appendix

Basic Tables Used in DES Algorithm

Table 31: *Permuted Choice 1*

Bit	0	1	2	3	4	5	6
1	57	49	41	33	25	17	9
8	1	58	50	42	34	26	18
15	10	2	59	51	43	35	27
22	19	11	3	60	52	44	36
29	63	55	47	39	31	23	15
36	7	62	54	46	38	30	22
43	14	6	61	53	45	37	29
50	21	13	5	28	20	12	4

Table 32: *Permuted Choice 2*

Bit	0	1	2	3	4	5
1	14	17	11	24	1	5
7	3	28	15	6	21	10
13	23	19	12	4	26	8
19	16	7	27	20	13	2
25	41	52	31	37	47	55
31	30	40	51	45	33	48
37	44	49	39	56	34	53
43	46	42	50	36	29	32

Table 33: *Subkey Rotation*

Round Number	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Number of bits to rotate	1	1	2	2	2	2	2	2	1	2	2	2	2	2	2

Table 34: *Initial Permutation*

Bit	0	1	2	3	4	5	6	7
1	58	50	42	34	26	18	10	2
9	60	52	44	36	28	20	12	4
17	62	54	46	38	30	22	14	6
25	64	56	48	40	32	24	16	8
33	57	49	41	33	25	17	9	1
41	59	51	43	35	27	19	11	3
49	61	53	45	37	29	21	13	5
57	63	55	47	39	31	23	15	7

Table 35: *Inverse Initial Permutation*

Bit	0	1	2	3	4	5	6	7
1	40	8	48	16	56	24	64	32
9	39	7	47	15	55	23	63	31
17	38	6	46	14	54	22	62	30
25	37	5	45	13	53	21	61	29
33	36	4	44	12	52	20	60	28
41	35	3	43	11	51	19	59	27
49	34	2	42	10	50	18	58	26
57	33	1	41	9	49	17	57	25

Table 36: *E-Bit Selection*

Bit	0	1	2	3	4	5
1	32	1	2	3	4	5
7	4	5	6	7	8	9
13	8	9	10	11	12	13
19	12	13	14	15	16	17
25	16	17	18	19	20	21
31	20	21	22	23	24	25
37	24	25	26	27	28	29
43	28	29	30	31	32	1

Table 37: *P Permutation*

Bit	0	1	2	3
1	16	7	20	21
5	29	12	28	17
9	1	15	23	26
13	5	18	31	10
17	2	8	24	14
21	32	27	3	9
25	19	13	30	6
29	22	11	4	25

Table 38: *S-Box 1*

Row/ Column	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	14	4	13	1	2	15	11	8	3	10	6	12	5	9	0	7
1	0	15	7	4	14	2	13	1	10	6	12	11	9	5	3	8
2	4	1	14	8	13	6	2	11	15	12	9	7	3	10	5	0
3	15	12	8	2	4	9	1	7	5	11	3	14	10	0	6	13

Table 39: *S-Box 2*

Row/ Column	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	15	1	8	14	6	11	3	4	9	7	2	13	12	0	5	10
1	3	13	4	7	15	2	8	14	12	0	1	10	6	9	11	5
2	0	14	7	11	10	4	13	1	5	8	12	6	9	3	2	15
3	13	8	10	1	3	15	4	2	11	6	7	12	0	5	14	9

Table 40: *S-Box 3*

Row/ Column	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	10	0	9	14	6	3	15	5	1	13	12	7	11	4	2	8
1	13	7	0	9	3	4	6	10	2	8	5	14	12	11	15	1
2	13	6	4	9	8	15	3	0	11	1	2	12	5	10	14	7
3	1	10	13	0	6	9	8	7	4	15	14	3	11	5	2	12

Table 41: *S-Box 4*

Row/ Column	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	7	13	14	3	0	6	9	10	1	2	8	5	11	12	4	15
1	13	8	11	5	6	15	0	3	4	7	2	12	1	10	14	9
2	10	6	9	0	12	11	7	13	15	1	3	14	5	2	8	4
3	3	15	0	6	10	1	13	8	9	4	5	11	12	7	2	14

Table 42: *S-Box 5*

Row/ Column	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	2	12	4	1	7	10	11	6	8	5	3	15	13	0	14	9
1	14	11	2	12	4	7	13	1	5	0	15	10	3	9	8	6
2	4	2	1	11	10	13	7	8	15	9	12	5	6	3	0	14
3	11	8	12	7	1	14	2	13	6	15	0	9	10	4	5	3

Table 43: *S-Box 6*

Row/ Column	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	12	1	10	15	9	2	6	8	0	13	3	4	14	7	5	11
1	10	15	4	2	7	12	9	5	6	1	13	14	0	11	3	8
2	9	14	15	5	2	8	12	3	7	0	4	10	1	13	11	6
3	4	3	2	12	9	5	15	10	11	14	1	7	6	0	8	13

Table 44: *S-Box 7*

Row/ Column	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	4	11	2	14	15	0	8	13	3	12	9	7	5	10	6	1
1	13	0	11	7	4	9	1	10	14	3	5	12	2	15	8	6
2	1	4	11	13	12	3	7	14	10	15	6	8	0	5	9	2
3	6	11	13	8	1	4	10	7	9	5	0	15	14	2	3	12

Table 45: *S-Box 8*

Row/ Column	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	13	2	8	4	6	15	11	1	10	9	3	14	5	0	12	7
1	1	15	13	8	10	3	7	4	12	5	6	11	0	14	9	2
2	7	11	4	1	9	12	14	2	0	6	10	13	15	3	5	8
3	2	1	14	7	4	10	8	13	15	12	9	0	3	5	6	11

Basic Tables Used in AES Algorithm

Table 46: *Rcon[256]*

Rcon[256] =

{0x8d, 0x01, 0x02, 0x04, 0x08, 0x10, 0x20, 0x40, 0x80, 0x1b, 0x36, 0x6c, 0xd8, 0xab, 0x4d, 0x9a,0x2f, 0x5e, 0xbc, 0x63, 0xc6, 0x97, 0x35, 0x6a, 0xd4, 0xb3, 0x7d, 0xfa, 0xef, 0xc5, 0x91, 0x39,0x72, 0xe4, 0xd3, 0xbd, 0x61, 0xc2, 0x9f, 0x25, 0x4a, 0x94, 0x33, 0x66, 0xcc, 0x83, 0x1d, 0x3a,0x74, 0xe8, 0xcb, 0x8d, 0x01, 0x02, 0x04, 0x08, 0x10, 0x20, 0x40, 0x80, 0x1b, 0x36, 0x6c, 0xd8,0xab, 0x4d, 0x9a, 0x2f, 0x5e, 0xbc, 0x63, 0xc6, 0x97, 0x35, 0x6a, 0xd4, 0xb3, 0x7d, 0xfa, 0xef,0xc5, 0x91, 0x39, 0x72, 0xe4, 0xd3, 0xbd, 0x61, 0xc2, 0x9f, 0x25, 0x4a, 0x94, 0x33, 0x66, 0xcc,0x83, 0x1d, 0x3a, 0x74, 0xe8, 0xcb, 0x8d, 0x01, 0x02, 0x04, 0x08, 0x10, 0x20, 0x40, 0x80, 0x1b,0x36, 0x6c, 0xd8, 0xab, 0x4d, 0x9a, 0x2f, 0x5e, 0xbc, 0x63, 0xc6, 0x97, 0x35, 0x6a, 0xd4, 0xb3,0x7d, 0xfa, 0xef, 0xc5, 0x91, 0x39, 0x72, 0xe4, 0xd3, 0xbd, 0x61, 0xc2, 0x9f, 0x25, 0x4a, 0x94,0x33, 0x66, 0xcc, 0x83, 0x1d, 0x3a, 0x74, 0xe8, 0xcb, 0x8d, 0x01, 0x02, 0x04, 0x08, 0x10, 0x20,0x40, 0x80, 0x1b, 0x36, 0x6c, 0xd8, 0xab, 0x4d, 0x9a, 0x2f, 0x5e, 0xbc, 0x63, 0xc6, 0x97, 0x35,0x6a, 0xd4, 0xb3, 0x7d, 0xfa, 0xef, 0xc5, 0x91, 0x39, 0x72, 0xe4, 0xd3, 0xbd, 0x61, 0xc2, 0x9f,0x25, 0x4a, 0x94, 0x33, 0x66, 0xcc, 0x83, 0x1d, 0x3a, 0x74, 0xe8, 0xcb, 0x8d, 0x01, 0x02, 0x04,0x08, 0x10, 0x20, 0x40, 0x80, 0x1b, 0x36, 0x6c, 0xd8, 0xab, 0x4d, 0x9a, 0x2f, 0x5e, 0xbc, 0x63,0xc6, 0x97, 0x35, 0x6a, 0xd4, 0xb3, 0x7d, 0xfa, 0xef, 0xc5, 0x91, 0x39, 0x72, 0xe4, 0xd3, 0xbd,0x61, 0xc2, 0x9f, 0x25, 0x4a, 0x94, 0x33, 0x66, 0xcc, 0x83, 0x1d, 0x3a, 0x74, 0xe8, 0xcb, 0x8d}

Table 47: *Rijndael S-Box*

	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
00	63	7c	77	7b	f2	6b	6f	c5	30	01	67	2b	fe	d7	ab	76
10	ca	82	c9	7d	fa	59	47	f0	ad	d4	a2	af	9c	a4	72	c0
20	b7	fd	93	26	36	3f	f7	cc	34	a5	e5	f1	71	d8	31	15
30	04	c7	23	c3	18	96	05	9a	07	12	80	e2	eb	27	b2	75
40	09	83	2c	1a	1b	6e	5a	a0	52	3b	d6	b3	29	e3	2f	84
50	53	d1	00	ed	20	fc	b1	5b	6a	cb	be	39	4a	4c	58	cf
60	d0	ef	aa	fb	43	4d	33	85	45	f9	02	7f	50	3c	9f	a8
70	51	a3	40	8f	92	9d	38	f5	bc	b6	da	21	10	ff	f3	d2
80	cd	0c	13	ec	5f	97	44	17	c4	a7	7e	3d	64	5d	19	73
90	60	81	4f	dc	22	2a	90	88	46	ee	b8	14	de	5e	0b	db
a0	e0	32	3a	0a	49	06	24	5c	c2	d3	ac	62	91	95	e4	79
b0	e7	c8	37	6d	8d	d5	4e	a9	6c	56	f4	ea	65	7a	ae	08
c0	ba	78	25	2e	1c	a6	b4	c6	e8	dd	74	1f	4b	bd	8b	8a
d0	70	3e	b5	66	48	03	f6	0e	61	35	57	b9	86	c1	1d	9e
e0	e1	f8	98	11	69	d9	8e	94	9b	1e	87	e9	ce	55	28	df
f0	8c	a1	89	0d	bf	e6	42	68	41	99	2d	0f	b0	54	bb	16

Table 48: *Rijndael N-Box*

x \ y	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
0		00	19	01	32	02	1a	c6	4b	c7	1b	68	33	ee	df	03
1	64	04	e0	0e	34	8d	81	ef	4c	71	08	c8	f8	69	1c	c1
2	7d	c2	1d	b5	f9	b9	27	6a	4d	e4	a6	72	9a	c9	09	78
3	65	2f	8a	05	21	0f	e1	24	12	f0	82	45	35	93	da	8e
4	96	8f	db	bd	36	d0	ce	94	13	5c	d2	f1	40	46	83	38
5	66	dd	fd	30	bf	06	8b	62	b3	25	e2	98	22	88	91	10
6	7e	6e	48	c3	a3	b6	1e	42	3a	6b	28	54	fa	85	3d	ba
7	2b	79	0a	15	9b	9f	5e	ca	4e	d4	ac	e5	f3	73	a7	57
8	af	58	a8	50	f4	ea	d6	74	4f	ae	e9	d5	e7	e6	ad	e8
9	2c	d7	75	7a	eb	16	0b	f5	59	cb	5f	b0	9c	a9	51	a0
a	7f	0c	f6	6f	17	c4	49	ec	d8	43	1f	2d	a4	76	7b	b7
b	cc	bb	3e	5a	fb	60	b1	86	3b	52	a1	6c	aa	55	29	9d
c	97	b2	87	90	61	be	dc	fc	bc	95	cf	cd	37	3f	5b	d1
d	53	39	84	3c	41	a2	6d	47	14	2a	9e	5d	56	f2	d3	ab
e	44	11	92	d9	23	20	2e	89	b4	7c	b8	26	77	99	e3	a5
f	67	4a	ed	de	c5	31	fe	18	0d	63	8c	80	c0	f7	70	07

The Rijndael N-Box contains the N values, such that {xy} = $\{03\}^{N}$ for an element {xy}.

Table 49: *Rijndael E-Box*

x \ y	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
0	01	03	05	0f	11	33	55	ff	1a	2e	72	96	a1	f8	13	35
1	5f	e1	38	48	d8	73	95	a4	f7	02	06	0a	1e	22	66	aa
2	e5	34	5c	e4	37	59	eb	26	6a	be	d9	70	90	ab	e6	31
3	53	f5	04	0c	14	3c	44	cc	4f	d1	68	b8	d3	6e	b2	cd
4	4c	d4	67	a9	e0	3b	4d	d7	62	a6	f1	08	18	28	78	88
5	83	9e	b9	d0	6b	bd	dc	7f	81	98	b3	ce	49	db	76	9a
6	b5	c4	57	f9	10	30	50	f0	0b	1d	27	69	bb	d6	61	a3
7	fe	19	2b	7d	87	92	ad	ec	2f	71	93	ae	e9	20	60	a0
8	fb	16	3a	4e	d2	6d	b7	c2	5d	e7	32	56	fa	15	3f	41
9	c3	5e	e2	3d	47	c9	40	c0	5b	ed	2c	74	9c	bf	da	75
a	9f	ba	d5	64	ac	ef	2a	7e	82	9d	bc	df	7a	8e	89	80
b	9b	b6	c1	58	e8	23	65	af	ea	25	6f	b1	c8	43	c5	54
c	fc	1f	21	63	a5	f4	07	09	1b	2d	77	99	b0	cb	46	ca
d	45	cf	4a	de	79	8b	86	91	a8	e3	3e	42	c6	51	f3	0e
e	12	36	5a	ee	29	7b	8d	8c	8f	8a	85	94	a7	f2	0d	17
f	39	4b	dd	7c	84	97	a2	fd	1c	24	6c	b4	c7	52	f6	01

The Rijndael E-Box contains the field element {E}, such that {E} = $\{03\}^{(xy)}$ given (xy)

For the field used in Rijndael, {03} is a generator that produces Table 48 and Table 49. Table 48 shows that {57} = $\{03\}^{(62)}$ and {83} = $\{03\}^{(50)}$, where the brackets on the exponent values identify them as hexadecimal numbers. This gives the product as {57} · {83} = $\{03\}^{(62)\,+\,(50)}$ and since (62) + (50) = (b2) in hexadecimal, Table 49 gives the resulting product as {c1}. These tables can also be used to find the inverse of a field element since the $g^{(X)}$ has an inverse represented by $g^{(ff)-(X)}$. Hence the element {af} = $\{03\}^{(b7)}$ has the inverse $g^{(ff)-(b7)} = g^{(48)}$ = {62}. All elements except {00} have inverses.

Table 50: *Rijndael Inverse S-Box*

	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**a**	**b**	**c**	**d**	**e**	**f**
00	52	09	6a	d5	30	36	a5	38	bf	40	a3	9e	81	f3	d7	fb
10	7c	e3	39	82	9b	2f	ff	87	34	8e	43	44	c4	de	e9	cb
20	54	7b	94	32	a6	c2	23	3d	ee	4c	95	0b	42	fa	c3	4e
30	08	2e	a1	66	28	d9	24	b2	76	5b	a2	49	6d	8b	d1	25
40	72	f8	f6	64	86	68	98	16	d4	a4	5c	cc	5d	65	b6	92
50	6c	70	48	50	fd	ed	b9	da	5e	15	46	57	a7	8d	9d	84
60	90	d8	ab	00	8c	bc	d3	0a	f7	e4	58	05	b8	b3	45	06
70	d0	2c	1e	8f	ca	3f	0f	02	c1	af	bd	03	01	13	8a	6b
80	3a	91	11	41	4f	67	dc	ea	97	f2	cf	ce	f0	b4	e6	73
90	96	ac	74	22	e7	ad	35	85	e2	f9	37	e8	1c	75	df	6e
a0	47	f1	1a	71	1d	29	c5	89	6f	b7	62	0e	aa	18	be	1b
b0	fc	56	3e	4b	c6	d2	79	20	9a	db	c0	fe	78	cd	5a	f4
c0	1f	dd	a8	33	88	07	c7	31	b1	12	10	59	27	80	ec	5f
d0	60	51	7f	a9	19	b5	4a	0d	2d	e5	7a	9f	93	c9	9c	ef
e0	a0	e0	3b	4d	ae	2a	f5	b0	c8	eb	bb	3c	83	53	99	61
f0	17	2b	04	7e	ba	77	d6	26	e1	69	14	63	55	21	0c	7d

Table 51: *InvMixColumns Multiplication Matrix*

e	b	d	9
9	e	b	d
d	9	e	b
b	d	9	e

Table 52: *ENS_2-Box*

x \ y	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
0	c6	f8	ee	f6	ff	d6	de	91	60	02	ce	56	e7	b5	4d	ec
1	8f	1f	89	fa	ef	b2	8e	fb	41	b3	5f	45	23	53	e4	9b
2	75	e1	3d	4c	6c	7e	f5	83	68	51	d1	f9	e2	ab	62	2a
3	08	95	46	9d	30	37	0a	2f	0e	24	1b	df	cd	4e	7f	ea
4	12	1d	58	34	36	dc	b4	5b	a4	76	b7	7d	52	dd	5e	13
5	a6	b9		c1	40	e3	79	b6	d4	8d	67	72	94	98	b0	85
6	bb	c5	4f	ed	86	9a	66	11	8a	e9	04	fe	a0	78	25	4b
7	a2	5d	80	05	3f	21	70	f1	63	77	af	42	20	e5	fd	bf
8	81	18	26	c3	be	35	88	2e	93	55	fc	7a	c8	ba	32	e6
9	c0	19	9e	a3	44	54	3b	0b	8c	c7	6b	28	a7	bc	16	ad
a	db	64	74	14	92	0c	48	b8	9f	bd	43	c4	39	31	d3	f2
b	d5	8b	6e	da	01	b1	9c	49	d8	ac	f3	cf	ca	f4	47	10
c	6f	f0	4a	5c	38	57	73	97	cb	a1	e8	3e	96	61	0d	0f
d	e0	7c	71	cc	90	06	f7	1c	c2	6a	ae	69	17	99	3a	27
e	d9	eb	2b	22	d2	a9	07	33	2d	3c	15	c9	87	aa	50	a5
f	03	59	09	1a	65	d7	84	d0	82	29	5a	1e	7b	a8	6d	2c

Table 53: *ENS_3-Box*

x \ y	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
0	a5	84	99	8d	0d	bd	b1	54	50	03	a9	7d	19	62	e6	9a
1	45	9d	40	87	15	eb	c9	0b	ec	67	fd	ea	bf	f7	96	5b
2	c2	1c	ae	6a	5a	41	02	4f	5c	f4	34	08	93	73	53	3f
3	0c	52	65	5e	28	a1	0f	b5	09	36	9b	3d	26	69	cd	9f
4	1b	9e	74	2e	2d	b2	ee	fb	f6	4d	61	ce	7b	3e	71	97
5	f5	68		2c	60	1f	c8	ed	be	46	d9	4b	de	d4	e8	4a
6	6b	2a	e5	16	c5	d7	55	94	cf	10	06	81	f0	44	ba	e3
7	f3	fe	c0	8a	ad	bc	48	04	df	c1	75	63	30	1a	0e	6d
8	4c	14	35	2f	e1	a2	cc	39	57	f2	82	47	ac	e7	2b	95
9	a0	98	d1	7f	66	7e	ab	83	ca	29	d3	3c	79	e2	1d	76
a	3b	56	4e	1e	db	0a	6c	e4	5d	6e	ef	a6	a8	a4	37	8b
b	32	43	59	b7	8c	64	d2	e0	b4	fa	07	25	af	8e	e9	18
c	d5	88	6f	72	24	f1	c7	51	23	7c	9c	21	dd	dc	86	85
d	90	42	c4	aa	d8	05	01	12	a3	5f	f9	d0	91	58	27	b9
e	38	13	b3	33	bb	70	89	a7	b6	22	92	20	49	ff	78	7a
f	8f	f8	80	17	da	31	c6	b8	c3	b0	77	11	cb	fc	d6	3a

Table 54: *Inverse ENS_E-Box*

x \ y	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
0	51	7e	1a	3a	3b	1f	ac	4b	20	ad	88	f5	4f	c5	26	b5
1	de	25	45	5d	c3	81	8d	6b	03	15	bf	95	d4	58	49	8e
2	75	f4	99	27	be	f0	c9	7d	63	e5	97	62	b1	bb	fe	f9
3	70	8f	94	52	ab	72	e3	66	b2	2f	86	d3	30	23	02	ed
4	8a	a7	f3	4e	65	06	d1	c4	34	a2	05	a4	0b	40	5e	bd
5	3e	96	dd	4d	91	71	04	60	19	d6	89	67	b0	07	e7	79
6	a1	7c	f8		09	32	1e	6c	fd	0f	3d	36	0a	68	9b	24
7	0c	93	b4	1b	80	61	5a	1c	e2	c0	3c	12	0e	f2	2d	14
8	57	af	ee	a3	f7	5c	44	5b	8b	cb	b6	b8	d7	42	13	84
9	85	d2	ae	c7	1d	dc	0d	77	2b	a9	11	47	a8	a0	56	22
a	87	d9	8c	98	a6	a5	da	3f	2c	50	6a	54	f6	90	2e	82
b	9f	69	6f	cf	c8	10	e8	db	cd	6e	ec	83	e6	aa	21	ef
c	ba	4a	ea	29	31	2a	c6	35	74	fc	e0	33	f1	41	7f	17
d	76	43	cc	e4	9e	4c	c1	46	9d	01	fa	fb	b3	92	e9	6d
e	9a	37	59	eb	ce	b7	e1	7a	9c	55	18	73	53	5f	df	78
f	ca	b9	38	c2	16	bc	28	ff	39	08	d8	64	7b	d5	48	d0

Table 55: *Inverse ENS_B-Box*

x \ y	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
0	50	53	c3	96	cb	f1	ab	93	55	f6	91	25	fc	d7	80	8f
1	49	67	98	e1	02	12	a3	c6	e7	95	eb	da	2d	d3	29	44
2	6a	78	6b	dd	b6	17	66	b4	18	82	60	45	e0	84	1c	94
3	58	19	87	b7	23	e2	57	2a	07	03	9a	a5	f2	b2	ba	5c
4	2b	92	f0	a1	cd	d5	1f	8a	9d	a0	32	75	39	aa	06	51
5	f9	3d	ae	46	b5	05	6f	ff	24	97	cc	77	bd	88	38	db
6	47	e9	c9		83	48	ac	4e	fb	56	1e	27	64	21	d1	3a
7	b1	0f	d2	9e	4f	a2	69	16	0a	e5	43	1d	0b	ad	b9	c8
8	85	4c	bb	fd	9f	bc	c5	34	76	dc	68	63	ca	10	40	20
9	7d	f8	11	6d	4b	f3	ec	d0	6c	99	fa	22	c4	1a	d8	ef
a	c7	c1	fe	36	cf	28	26	a4	e4	0d	9b	62	c2	e8	5e	f5
b	be	7c	a9	b3	3b	a7	6e	7b	09	f4	01	a8	65	7e	08	e6
c	d9	ce	d4	d6	af	31	30	c0	37	a6	b0	15	4a	f7	0e	2f
d	8d	4d	54	df	e3	1b	b8	7f	04	5d	73	2e	5a	52	33	13
e	8c	7a	8e	89	ee	35	ed	3c	59	3f	79	bf	ea	5b	14	86
f	81	3e	2c	5f	72	0c	8b	41	71	de	9c	90	61	70	74	42

Table 56: *Inverse ENS_D-Box*

x \ y	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
0	a7	65	a4	5e	6b	45	58	03	fa	6d	76	4c	d7	cb	44	a3
1	5a	1b	0e	c0	75	f0	97	f9	5f	9c	7a	59	83	21	69	c8
2	89	79	3e	71	4f	ad	ac	3a	4a	31	33	7f	77	ae	a0	2b
3	68	fd	6c	f8	d3	02	8f	ab	28	c2	7b	08	87	a5	6a	82
4	1c	b4	f2	e2	f4	be	62	fe	53	55	e1	eb	ec	ef	9f	10
5	8a	06	05	bd	8d	5d	d4	15	fb	e9	43	9e	42	8b	5b	ee
6	0a	0f	1e		86	ed	70	72	ff	38	d5	39	d9	a6	54	2e
7	67	e7	96	91	c5	20	4b	1a	ba	2a	e0	17	0d	c7	a8	a9
8	19	07	dd	60	26	f5	3b	7e	29	c6	fc	f1	dc	85	22	11
9	24	3d	32	a1	2f	30	52	e3	16	b9	48	64	8c	3f	2c	90
a	4e	d1	a2	0b	81	de	8e	bf	9d	92	cc	46	13	b8	f7	af
b	80	93	2d	12	99	7d	63	bb	78	18	b7	9a	6e	e6	cf	e8
c	9b	36	09	7c	b2	23	94	66	bc	ca	d0	d8	98	da	50	f6
d	d6	b0	4d	04	b5	88	1f	51	ea	35	74	41	1d	d2	56	47
e	61	0c	14	3c	27	c9	e5	b1	df	73	ce	37	cd	aa	6f	db
f	f3	c4	34	40	c3	25	49	95	01	b3	e4	c1	84	b6	5c	57

Table 57: *Inverse ENS_9-Box*

x \ y	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
0	f4	41	17	27	ab	9d	fa	e3	30	76	cc	02	e5	2a	35	62
1	b1	ba	ea	fe	2f	4c	46	d3	8f	92	6d	52	be	74	e0	c9
2	c2	8e	58	b9	e1	88	20	ce	df	1a	51	53	64	6b	81	08
3	48	45	de	7b	73	4b	1f	55	eb	b5	c5	37	28	bf	03	16
4	cf	79	07	69	da	05	34	a6	2e	f3	8a	f6	83	60	71	6e
5	21	dd	3e	e6	54	c4	06	50	98	bd	40	d9	e8	89	19	c8
6	7c	42	84		80	2b	11	5a	0e	85	ae	2d	0f	5c	5b	36
7	0a	57	ee	9b	c0	dc	77	12	93	a0	22	1b	09	8b	b6	1e
8	f1	75	99	7f	01	72	66	fb	43	23	ed	e4	31	63	97	c6
9	4a	bb	f9	29	9e	b2	86	c1	b3	70	94	e9	fc	f0	7d	33
a	49	38	ca	d4	f5	7a	b7	ad	3a	78	5f	7e	8d	d8	39	c3
b	5d	d0	d5	25	ac	18	9c	3b	26	59	9a	4f	95	ff	bc	15
c	e7	6f	9f	b0	a4	3f	a5	a2	4e	82	90	a7	04	ec	cd	91
d	4d	ef	aa	96	d1	6a	2c	65	5e	8c	87	0b	67	db	10	d6
e	d7	a1	f8	13	a9	61	1c	47	d2	f2	14	c7	f7	fd	3d	44
f	af	68	24	a3	1d	e2	3c	0d	a8	0c	b4	56	cb	32	6c	b8

Table 58: *EN_E-Box*

x \ y	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
0		0e	1c	12	38	36	24	2a	70	7e	6c	62	48	46	54	5a
1	e0	ee	fc	f2	d8	d6	c4	ca	90	9e	8c	82	a8	a6	b4	ba
2	db	d5	c7	c9	e3	ed	ff	f1	ab	a5	b7	b9	93	9d	8f	81
3	3b	35	27	29	03	0d	1f	11	4b	45	57	59	73	7d	6f	61
4	ad	a3	b1	bf	95	9b	89	87	dd	d3	c1	cf	e5	eb	f9	f7
5	4d	43	51	5f	75	7b	69	67	3d	33	21	2f	05	0b	19	17
6	76	78	6a	64	4e	40	52	5c	06	08	1a	14	3e	30	22	2c
7	96	98	8a	84	ae	a0	b2	bc	e6	e8	fa	f4	de	d0	c2	cc
8	41	4f	5d	53	79	77	65	6b	31	3f	2d	23	09	07	15	1b
9	a1	af	bd	b3	99	97	85	8b	d1	df	cd	c3	e9	e7	f5	fb
a	9a	94	86	88	a2	ac	be	b0	ea	e4	f6	f8	d2	dc	ce	c0
b	7a	74	66	68	42	4c	5e	50	0a	04	16	18	32	3c	2e	20
c	ec	e2	f0	fe	d4	da	c8	c6	9c	92	80	8e	a4	aa	b8	b6
d	0c	02	10	1e	34	3a	28	26	7c	72	60	6e	44	4a	58	56
e	37	39	2b	25	0f	01	13	1d	47	49	5b	55	7f	71	63	6d
f	d7	d9	cb	c5	ef	e1	f3	fd	a7	a9	bb	b5	9f	91	83	8d

Table 59: *EN_B-Box*

x \ y	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
0		0b	16	1d	2c	27	3a	31	58	53	4e	45	74	7f	62	69
1	b0	bb	a6	ad	9c	97	8a	81	e8	e3	fe	f5	c4	cf	d2	d9
2	7b	70	6d	66	57	5c	41	4a	23	28	35	3e	0f	04	19	12
3	cb	c0	dd	d6	e7	ec	f1	fa	93	98	85	8e	bf	b4	a9	a2
4	f6	fd	e0	eb	da	d1	cc	c7	ae	a5	b8	b3	82	89	94	9f
5	46	4d	50	5b	6a	61	7c	77	1e	15	08	03	32	39	24	2f
6	8d	86	9b	90	a1	aa	b7	bc	d5	de	c3	c8	f9	f2	ef	e4
7	3d	36	2b	20	11	1a	07	0c	65	6e	73	78	49	42	5f	54
8	f7	fc	e1	ea	db	d0	cd	c6	af	a4	b9	b2	83	88	95	9e
9	47	4c	51	5a	6b	60	7d	76	1f	14	09	02	33	38	25	2e
a	8c	87	9a	91	a0	ab	b6	bd	d4	df	c2	c9	f8	f3	ee	e5
b	3c	37	2a	21	10	1b	06	0d	64	6f	72	79	48	43	5e	55
c	01	0a	17	1c	2d	26	3b	30	59	52	4f	44	75	7e	63	68
d	b1	ba	a7	ac	9d	96	8b	80	e9	e2	ff	f4	c5	ce	d3	d8
e	7a	71	6c	67	56	5d	40	4b	22	29	34	3f	0e	05	18	13
f	ca	c1	dc	d7	e6	ed	f0	fb	92	99	84	8f	be	b5	a8	a3

Table 60: *EN_D-Box*

x \ y	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
0		0d	1a	17	34	39	2e	23	68	65	72	7f	5c	51	46	4b
1	d0	dd	ca	c7	e4	e9	fe	f3	b8	b5	a2	af	8c	81	96	9b
2	bb	b6	a1	ac	8f	82	95	98	d3	de	c9	c4	e7	ea	fd	f0
3	6b	66	71	7c	5f	52	45	48	03	0e	19	14	37	3a	2d	20
4	6d	60	77	7a	59	54	43	4e	05	08	1f	12	31	3c	2b	26
5	bd	b0	a7	aa	89	84	93	9e	d5	d8	cf	c2	e1	ec	fb	f6
6	d6	db	cc	c1	e2	ef	f8	f5	be	b3	a4	a9	8a	87	90	9d
7	06	0b	1c	11	32	3f	28	25	6e	63	74	79	5a	57	40	4d
8	da	d7	c0	cd	ee	e3	f4	f9	b2	bf	a8	a5	86	8b	9c	91
9	0a	07	10	1d	3e	33	24	29	62	6f	78	75	56	5b	4c	41
a	61	6c	7b	76	55	58	4f	42	09	04	13	1e	3d	30	27	2a
b	b1	bc	ab	a6	85	88	9f	92	d9	d4	c3	ce	ed	e0	f7	fa
c	b7	ba	ad	a0	83	8e	99	94	df	d2	c5	c8	eb	e6	f1	fc
d	67	6a	7d	70	53	5e	49	44	0f	02	15	18	3b	36	21	2c
e	0c	01	16	1b	38	35	22	2f	64	69	7e	73	50	5d	4a	47
f	dc	d1	c6	cb	e8	e5	f2	ff	b4	b9	ae	a3	80	8d	9a	97

Table 61: *EN_9-Box*

x \ y	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
0		09	12	1b	24	2d	36	3f	48	41	5a	53	6c	65	7e	77
1	90	99	82	8b	b4	bd	a6	af	d8	d1	ca	c3	fc	f5	ee	e7
2	3b	32	29	20	1f	16	0d	04	73	7a	61	68	57	5e	45	4c
3	ab	a2	b9	b0	8f	86	9d	94	e3	ea	f1	f8	c7	ce	d5	dc
4	76	7f	64	6d	52	5b	40	49	3e	37	2c	25	1a	13	08	01
5	e6	ef	f4	fd	c2	cb	d0	d9	ae	a7	bc	b5	8a	83	98	91
6	4d	44	5f	56	69	60	7b	72	05	0c	17	1e	21	28	33	3a
7	dd	d4	cf	c6	f9	f0	eb	e2	95	9c	87	8e	b1	b8	a3	aa
8	ec	e5	fe	f7	c8	c1	da	d3	a4	ad	b6	bf	80	89	92	9b
9	7c	75	6e	67	58	51	4a	43	34	3d	26	2f	10	19	02	0b
a	d7	de	c5	cc	f3	fa	e1	e8	9f	96	8d	84	bb	b2	a9	a0
b	47	4e	55	5c	63	6a	71	78	0f	06	1d	14	2b	22	39	30
c	9a	93	88	81	be	b7	ac	a5	d2	db	c0	c9	f6	ff	e4	ed
d	0a	03	18	11	2e	27	3c	35	42	4b	50	59	66	6f	74	7d
e	a1	a8	b3	ba	85	8c	97	9e	e9	e0	fb	f2	cd	c4	df	d6
f	31	38	23	2a	15	1c	07	0e	79	70	6b	62	5d	54	4f	46

List of Figures

List of Tables

References

[1] Asic World. "*Verilog Synthesis Tutorial*", asic-world.com. [Online]. Available: http://www.asic-world.com/verilog/synthesis.html [Accessed: Mar. 23, 2012].

[2] COSIC. "*Test Vectors*", cosic.esat.kuleuven.be. [Online]. Available: https://www.cosic.esat.kuleuven.be/nessie/testvectors [Accessed: Apr. 5, 2012].

[3] M. Lucky (2008, Dec.). "*AES Encryption and CAST's AES IP Cores*". Cast [Online]. Available: http://www.cast-inc.com/ip-cores/encryption/cast-AES-IP-overview.pdf. [Accessed: Apr. 10, 2012].

[4] D. Harris (2000, Sep. 9). "*Structural Design with Verilog*". Massachusetts Institute of Technology [Online]. Available: http://www.mit.bme.hu/system/files/oktatas/targyak/8136/verilog.pdf. [Accessed: Mar. 24, 2012].

[5] R. Lorencz. "*New Algorithm for Classical Modular Inverse*". Czech Technical University in Prague [Online]. Available: http://users.fit.cvut.cz/~lorencz/clanky/1_New_Alg_Class_Inv.pdf. [Accessed: Jul. 2, 2012].

[6] Chia-Long WU, Der-Chyuan LOU and Te-Jen CHANG (2005, Apr.). "*An Efficient Montgomery Exponentiation Algorithm for Cryptographic Applications*". VU Matematikos ir Informatikos Institutas [Online]. Available: http://www.mii.lt/Informatica/pdf/INFO600.pdf. [Accessed: Jul. 24, 2012].

[7] EECS. "*AMBA 3 AHB-Lite Protocol*", eecs.umich.edu. [Online]. Available: http://www.eecs.umich.edu/eecs/courses/eecs373/readings/ARM_IHI0033A_AMBA_AHB-Lite_SPEC.pdf [Accessed: Apr. 1, 2012].

[8] Tropical Software. "*DES Encryption*", tropsoft.com. [Online]. Available: http://www.tropsoft.com/strongenc/des.htm [Accessed: Mar. 28, 2012].

[9] Dr. B. Gladman (2001, Mar. 3). "*A Specification for Rijndael, the AES Algorithm*". University of Sussex [Online]. Available: http://www.comms.engg.susx.ac.uk/fft/crypto/aesspec.pdf. [Accessed: Apr. 2, 2012].

[10] R. Ranjan and I. Poonguzhali. "*VLSI Implementation of IDEA Encryption Algorithm*". Technology Information Forecasting and Assessment Council [Online]. Available: http://tifac.velammal.org/CoMPC/articles/20.pdf. [Accessed: Jun. 5, 2012].

[11] C. Koc and T. Acar (1998, Apr.). "*Montgomery Multiplication in GF(2^k)*". University of Western Ontario [Online]. Available: http://www.csd.uwo.ca/~eschost/Exam/Koc.pdf. [Accessed: Aug. 4, 2012].

[12] A. Daly and W. Marnane (2002, Feb.). "*Efficient Architectures for implementing Montgomery Modular Multiplication and RSA Modular Exponentiation on Reconfigurable Logic*". University of York [Online]. Available: http://www.cs.york.ac.uk/rts/docs/SIGDA-Compendium-1994-2004/papers/2002/fpga02/pdffiles/2_2.pdf. [Accessed: Aug. 4, 2012].

[13] A. Dhir (2000, Mar. 9). "*Data Encryption using DES/Triple-DES Functionality in Spartan-II FPGAs*". Xilinx [Online]. Available: http://www.xilinx.com/support/documentation/white_papers/wp115.pdf. [Accessed: Mar. 3, 2012].

[14] B. Schneier, *Applied Cryptography*. [E-book] Available: http://www.cse.iitk.ac.in/users/anuag/crypto.pdf. [Accessed Apr. 15, 2012]

[15] A. Canteaut, F. Levy-dit-Vehel and G. Norton. "*Modern cryptology*". Inria [Online]. Available: https://www.rocq.inria.fr/secret/Anne.Canteaut/modern-crypto.pdf. [Accessed: Sep. 14, 2012].

[16] University of York. "*IEEE referencing style*", york.ac.uk. [Online]. Available: http://www.york.ac.uk/integrity/ieee.html [Accessed: Oct. 2, 2012].

[17] Wikipedia. "*Cryptography*", Wikipedia.org. [Online]. Available: http://en.wikipedia.org/wiki/Cryptography [Last Modified: 12 October 2012, 01:01]

[18] Wikipedia. "*Advanced Encryption Standard process*", Wikipedia.org. [Online]. Available: http://en.wikipedia.org/wiki/Advanced_Encryption_Standard_process [Last Modified: 3 October 2012, 13:44]

[19] Wikipedia. "*Data Encryption Standard*", Wikipedia.org. [Online]. Available: http://en.wikipedia.org/wiki/Data_Encryption_Standard [Last Modified: 10 October 2012, 14:57]

[20] Wikipedia. "*International Data Encryption Algorithm*", Wikipedia.org. [Online]. Available: http://en.wikipedia.org/wiki/International_Data_Encryption_Algorithm [Last Modified: 12 October 2012, 20:15]

[21] Wikipedia. "*RSA (algorithm)*", Wikipedia.org. [Online]. Available: http://en.wikipedia.org/wiki/RSA_(algorithm) [Last Modified: 13 October 2012, 21:21]

[22] R. Zimmermann, "Efficient VLSI Implementation of Modulo ($2^n \pm 1$) Addition and Multiplication", presented at the 14th IEEE Symposium on Computer Arithmetic (ARITH 14), Adelaide, Australia, April, 1999

[23] *AMBA™ Specification (Rev 2.0)*, ARM, 1999

[24] N. Moshopoulos, "Κρυπτογραφία Δημόσιου και Ιδιωτικού Κλειδιού. Αποδοτική Υλοποίηση Αλγόριθμων σε VLSI", Ph.D. dissertation, School of Electrical and Computer Engineering, National Technical University of Athens, 2001

[25] T. Padmanabhan and B. Bala Tripura Sundari, *Design Through Verilog HDL*. Piscataway: IEEE Press, 2004.

[26] *DE4 User Manual*, Terasic Technologies Inc., 2010.