Secure Computing
on Reconfigurable Systems

Secure Computing
on Reconfigurable Systems

PROEFSCHRIFT

ter verkrijging van de graad van doctor
aan de Technische Universiteit Delft,
op gezag van de Rector Magnificus Prof.dr.ir. J.T. Fokkema,
voorzitter van het College voor Promoties,
in het openbaar te verdedigen

op dinsdag 11 december 2007 om 15:00 uur

door

Ricardo Jorge FERNANDES CHAVES

elektrotechnisch ingenieur
Technical University of Lisbon
geboren te Lisboa, Portugal

Dit proefschrift is goedgekeurd door de promotors:
Prof. dr. K. Goossens
Prof. dr. L. Sousa

Samenstelling promotiecommissie:

Rector Magnificus, voorzitter	Technische Universiteit Delft
Prof. dr. K. G. W. Goossens, promotor	Technische Universiteit Delft
Prof. dr. L. Sousa, promotor	Universidade Técnica de Lisboa
Prof. dr. L. Silveira	Universidade Técnica de Lisboa
Prof. dr. A. Ferrari	Universidade de Aveiro
Prof. dr. K. Beenakker	Technische Universiteit Delft
Prof. dr. J. Lubbe	Technische Universiteit Delft
Prof. dr. P. French, reservelid	Technische Universiteit Delft

My advisor Professor Stamatis Vassiliadis has provided substantial guidance and support in the preparation of this thesis.

Universidade Técnica de Lisboa, Instituto Superior Técnico made important contributions to the work described in this dissertation. Financial support was provided by the Portuguese Foundation for Science and Technology.

ISBN 978-90-807957-5-4

Subject headings: Secure/trusted computing, reconfigurable systems, cryptography.

Printed in The Netherlands

in memoriam Professor Stamatis Vassiliadis

Secure Computing
on Reconfigurable Systems

Abstract

This thesis proposes a Secure Computing Module (SCM) for reconfigurable computing systems. SC provides a protected and reliable computational environment, where data security and protection against malicious attacks to the system is assured. SC is strongly based on encryption algorithms and on the attestation of the executed functions. The use of SC on reconfigurable devices has the advantage of being highly adaptable to the application and the user requirements, while providing high performances. Moreover, it is adaptable to new algorithms, protocols, and threats. In this dissertation, high performance cryptographic units for symmetric encryption and hash functions, were designed in order to achieve a high performance SCM. Implementations results, in particular for the AES algorithm, suggest improvements of more than 500% in terms of Throughput per Slice compared to related art, with absolute throughputs of up to 34Gbit/s on a Virtex II Pro FPGA. A method to attest dynamically reconfigured hardware structures is also proposed. In addition, this method does not penalize the performance of the SCM. The presented attestation mechanism allows the configuration bitstreams to be stored in unsecured locations, for example on an external memory or even on the internet, without posing a security threat. Experimental results obtained by implementing the proposed SCM on a Virtex II Pro FPGA suggest speedups up to 750 times, compared with software implemented algorithms, achieving throughputs above 1Gbit/s at low area cost. Overall, this dissertation demonstrates the applicability and identifies the main advantages of implementing SC on reconfigurable systems.

Key words: Secure Computing, hardware attestation, reconfigurable systems, FPGA

Secure Computing
on Reconfigurable Systems

Resumo

Esta tese propõe um módulo para Computação Segura (*Secure Computing* - SC) em sistemas de computação reconfigurável. A SC proporciona um ambiente de computação protegido e de confiança, em que a segurança dos dados e a protecção contra ataques maliciosos ao sistema é assegurada. A SC é fundamentalmente baseada em algoritmos de cifragem e na atestação das funções executadas. A utilização de SC em dispositivos reconfiguráveis tem a vantagem de se adaptar à aplicação e aos requisitos do utilizador, enquanto permite elevados desempenhos, sendo também adaptável a novos algoritmos de cifragem, protocolos e ameaças. Nesta dissertação, unidades criptográficas de elevado desempenho para cifragem simétrica e funções de dispersão foram projectadas de forma a obter um módulo de eficiente de SC. Resultados da implementação, em particular para o algoritmo AES, sugerem melhorias superiores a 500%, no que respeita à razão do ritmo de saída pelos blocos reconfiguráveis usados, comparativamente ao estado da arte; foram obtidos ritmos de saída até 34Gbit/s numa FPGA Virtex II Pro. Um novo método é também proposto para a atestação de estruturas computacionais dinamicamente reconfiguradas, sem que o desempenho do módulo SC seja comprometido. Este método permite que o *bitstream* de configuração possa ser armazenado em locais não seguros, por exemplo numa memória externa ou mesmo numa localização remota acessível pela internet, sem que isso constitua uma falha de segurança. Resultados experimentais obtidos com a implementação do modulo SC numa FPGA Virtex II Pro sugerem acelerações de processamento até 750 vezes, comparativamente com as implementações dos algoritmos em *software*, obtendo-se ritmos de saída superiores a 1Gbit/s com uma área de circuito reduzida. Pode-se dizer que esta tese mostra a aplicabilidade do conceito SC a sistemas reconfiguráveis, verificando-se experimentalmente as suas principais vantagens, com base em implementações em FPGA.

Palavras chaves: Secure Computing, atestação de *hardware*, sistemas reconfiguráveis, FPGA

Secure Computing
on Reconfigurable Systems

Samenvatting

Dit proefschrift introduceert een *Secure Computing Module* (SCM) voor herconfigureerbare rekensystemen. SC biedt een beschermde en betrouwbare rekenomgeving, die dataveiligheid en bescherming tegen aanvallen garandeert. SC is in sterke mate gebaseerd op het gebruikvan versleutelalgoritmes en de validatie van de uitgevoerde functionaliteit. Het voordeel van het gebruik van SC in herconfigureerbare apparaten is dat het gemakkelijk aangepast kan worden aan toepassing en gebuikerseisen, terwijl er toch een hoge snelheid geboden kan worden. Bovendien is het verder aan te passen aan nieuwe algoritmen, protocollen en dreigingen. In dit proefschrift worden een aantal snelle cryptografische eenheden voor symmetrische encryptie en *hash* functies ontworpen die nodig zijn om een snelle SCM te verkrijgen. Implementatie resultaten suggereren verbeteringen van meer dan 500% in *Throughput per Slice* ten opzichte van gerelateerd werk, met doorvoersnelheden tot 34Gbit/s in een Virtex II Pro FPGA. Daarnaast wordt er een nieuwe methode gepresenteerd om dynamisch geherconfigureerde hardwarestructuren te valideren zonder nadelige gevolgen voor de snelheid van de SCM. Hierdoor kunnen configuratiebitstromen opgeslagen worden op onveilige lokaties, bijvoorbeeld in extern geheugen of zelfs op het internet, zonder dat dit een veiligheidsrisico vormt. Experimentele evaluatie van een implementatie van de SCM in een Virtex II Pro FPGA leidde tot een maximale verbetering van een factor 750 ten opzichte van algoritmes geïmplementeerd in software; de implementatie haalde doorvoersnelheden van meer dan 1Gbit/s met een kleinere aanspraak op de beschikbare bronnen. Kortom, dit proefschrift illustreert de toepassing en de voordelen van de implementatie van SC in herconfigureerbare systemen.

Sleutelwoorden:*Secure Computing*, hardware validatie, herconfigureerbare systemen, FPGA

Acknowledgements

First and foremost I would like to show my profound gratitude to Prof. Stamatis Vassiliadis and Prof. Leonel Sousa, for their permanent and dedicated support throughout the development of this thesis and myself. Notwithstanding, I leave this "thank you" with a feeling of sorrow for not having enjoyed and grown more with the company, friendship, and guidance of the paragon that was Professor Stamatis Vassiliadis.

Also a special "thank you" to Dr. Georgi Kuzmanov, whose contribution to the development of this work was of truth importance. Even though he was officially only a colleague, he acted as my co-advisor, guiding and helping me to improve the quality of my work.

I would like to thank Dr. Georgi Gaydadjiev and Dr. Koen Bertels for all their support; especially to Dr. Gaydadjiev for the endeavor that was the establishment of the mix PhD contract between TULisbon and TUDelft. "Thank you" to Prof. Kees Goossens, for a thoughtful and constructive guidance as my TUDelft promoter in the last steps of my thesis.

I would like to thank Frederico and my two paranymphs, Marisa and Rui, for checking the text of this dissertation and everything else. "Thank you" also to Cathal for the Dutch translation of the abstract and the propositions. Also, "thank you" to my office mates from SiPS and CE lab for their friendship, help, the enlightened discussions, and all other conversations. "Thank you" to the CE lab and the SiPS group for this second home. A note of appreciation to the Portuguese Foundation for Science and Technology, for the financial support for this PhD work. Also, a "thank you" for everything else not mentioned and to all those not spoken off.

And finally, but certainly not least, a very warm and tender "thank you" to my fader, for all his love and support throughout this venture.

Ricardo Chaves Delft, The Netherlands, 2007

Contents

List of Tables

List of Figures

List of Acronyms

AES	Advanced Encryption Standard
ASIC	Application-Specific Integrated Circuit
BRAM	embedded Block RAM
CBC	Cipher-Block Chaining
CCU	Custom Computing Unit
CLB	Configurable Logic Block
CFB	Cipher FeedBack
CrCU	Cryptographic Computational Unit
CRT	Chinese Remainder Theorem
DES	Data Encryption Standard
DM	Digest Message
DPA	Differential Power Analysis
DS	Digital Signature
DSA	Digital Signature Algorithm
ECB	Electronic CodeBook
EPROM	Erasable Programmable ROM
FIFO	First In First Out buffer
FPGA	Field Programmable Gate Array
GPP	General Purpose Processors
HMAC	keyed-Hash Message Authentication Code
I/O	Input/Output
ICAP	Internal Configuration Access Port
IV	Initialization Vector
MD5	Message-Digest algorithm 5
OFB	Output FeedBack
RAM	Random access memory
RFID	Radio-Frequency IDentification
RNS	Residue Number System
ROM	Read Only memory

RSA	R. Rivest, A. Shamir, and L. Adleman asymmetrical encryption
SC	Secure Computing
SCM	Secure Computing Module
SHA	Secure Hash Algorithm
TC	Trusted Computing
TCAP	Trusted Computing Alliance Platform
TCG	Trusted Computing Group
TPM	Trusted Platform Module
XREG	eXchange REGister

Secure: able to avoid being harmed by any risk, danger, or threat.

Chapter 1

Introduction

Contents

With the vast expansion of electronic systems and digital information use, an increasing demand for secure and reliable computation environments arose. To deal with this demand several mechanisms are being developed and employed in most computational and communications systems. These mechanisms go from the protocol level to the architectural level, passing also by the use of new more and efficient security algorithms.

One of the most recent proposals with a large dissemination in the industrial and domestic market, is the Trusted Platform Module (TPM), used to implement the concept of trusted computing. This Trusted Computing concept is supported by the existence of hardware and software that allow the implementation of a set of features proposed by the Trusted Computing Group (TCG) [2, 3]. While the software part of Trusted Computing can be more or less easily used and adapted to the user/system's needs, the existing hardware support is rather closed, unadaptable, and controlled by a limited group of chip manufactures; this is due to security issues and adaptation constrains imposed by ASIC implementations.

Rather than Trusted Computing, hardware is developed in this thesis to support the Secure Computing approach on reconfigurable devices. In this Secure Computing perspective, the system and its usage is not limited to an authorized user. Moreover, the users can have control over the whole system, if required. This additional degree of freedom given to the user comes at the expense of the loss of trustworthiness in remote computation. This however, adds flexibility and adaptability to the Secure Computing Module (SCM) proposed in this thesis and implemented on reconfigurable devices. The proposed Secure Computing Module will allow a significant increase in the reliability and security of systems where the TPM cannot be efficiently used, namely on reconfigurable and dedicated structures. At the same time, the reconfigurable and customizable structures of the Secure Computing Module, will allow for a faster adaptation to new protocols, algorithms, and newly developed attacks.

This chapter presents an overview of the research developed in this thesis. Section 1.1 presents the background that led to the research work presented in this thesis. An introductory overview of the research developed is also presented in this section. In Section 1.2 the conceptual architecture for the proposed Secure Computing Module on reconfigurable computing devices is illustrated. This introductory chapter is concluded with the discussion of the thesis objectives and an overview on how the developed research work is structured and presented.

1.1 Background and Proposed Work

The definition of Trust states “firm reliance on the integrity; or the condition and resulting obligation of having confidence placed in one” contrasting with the definition of Secure which is defined as “able to avoid being harmed by any risk, danger or threat.” According to these definitions, trusted computing can be interpreted as the ability of having computational systems that are reliable and can maintain computational integrity, even when hardware degradation occurs. This will become an important issue when the technology reaches a point where system degradation will occur with a significantly higher probability [4–6]. These failures can be minimized by the use of self checking designs and the use of redundant logic, capable of functionally replacing the damaged part of the circuit.

The work presented in this thesis is focused on the other component of the definition of trusted computing, which is related with security: the capability of assuring confidence in the computational system that is being used. With the proliferation of digital data usage, the potential for violation of user’s privacy and data coherence significantly grows. Either remotely, with the use of programs developed to examine or modify the existing data and the systems usage e.g. virus and worms; or locally, through the monitoring of the systems behavior, e.g. printing a document from an unauthorized computer; or through physical attacks, e.g. by observing the power consumption or reading the data stored in memory.

With these issues in mind, the major software and hardware manufactures created the Trusted Computing Alliance Platform to normalize and catalyze the use of security systems, in order to achieve more secure and trustworthy computational systems.

1.1.1 Trusted Computing

This Trusted Computing Alliance Platform (TCAP), a consortium formed by Microsoft [7], Intel [8, 9], IBM, AMD [7], Sun Microsystems, HP, among others, also called Trusted Computing Group (TCG), have established a set of features to be used in future generation of computers, providing new standard for trusted computing [2, 3, 10]. These new capabilities were devised to be integrated at hardware and software application levels.

This group developed the Trusted Platform Module (TPM), which provides hardware acceleration for the proposed features and methods, namely: (i) Secure Input/Output (I/O); (ii) Memory curtaining; (iii) Sealed storage; (iv) Remote attestation; (v) Endorsement Key.

Secure input and output: The secure I/O feature consists on the validation of the received data by using a checksum approach to verify that the software used to generate the I/O has not been tampered with; at a higher level, data encryption can be performed to the data being transmitted to and from a peripheral device, in order to assure the secure I/O. An example of such an attack would be a malicious entity trying to snoop the communication between the computer and a credit card reading device.

Memory curtaining: Memory curtaining consists in only allowing access to a memory region to specific software applications, thus preventing other applications, e.g. virus, of access to critical data that can be misused, even if the malicious application took control of Operating System (OS). With this mechanism, security keys and other critical data can be safely stored. Only an authenticated application or user is able to access these data. For example if the OS is corrupted, the user's critical data, like private passwords/keys, will still remain secure, since not even the OS is able to access them without the proper authentication.

Sealed storage: Sealed storage consists in storing encrypted data into memory. The key used to encrypt the data is generated as a combination of the software application and/or machine hardware, which means that only a given combination of software and hardware is capable of correctly accessing the data stored in memory. This mechanism protects the users information of being read by a different application (or an adulterated version of the software) or from being read by an unauthorized machine. A typical example of sealed storage is the protection of a file in a hard drive; if a hard drive is stolen, a different computer should not be able to read the data.

Remote attestation: With remote attestation the software or a combination of software and hardware can be authenticated, generating a digital signature which depends on the software and the machine being used. This allows changes to the user's computer to be detected by an authorized challenger. Digital Signature (DS) algorithms are used to assure a remote recipient that the data was constructed by a non forged, cryptographically identified, trusted application.

The use of this feature allows for example for an internet banking system to only allow access to the service if the used internet browser has not been tampered with.

Endorsement Key: The Endorsement Key is typically a 2048 bit RSA key pair used in the identification of a TPM chip. This key pair is randomly created at manufacture time. This key pair is only known inside the chip and cannot be changed. No one has knowledge of the private Endorsement Key, not even the user.

The attestation of the TPM is performed in an identical manner as the remote attestation; the challenger sends a random number that has to be encrypted in the TPM with the private Endorsement Key. Thus, the TPM is validated only if the challenger is able to properly decrypt the random number, with the public Endorsement Key of that TPM chip.

Drawbacks of the ASIC Implemented TPM

The use of a TPM chip supplies additional security mechanisms to the current computational systems. However, both the TPM chip and the Trusted Computing concept have limitations in applications that require adaptable or customizable properties.

Regarding the TPM chip, the use of a static chip, without any type of adaptation capability, causes it to become obsolete as new protocols and algorithms are created, making all the system also obsolete. For example, only in the recent revision of the trusted computing group (1.2) has the AES encryption been included, becoming a mandatory algorithm [11, 12]. This lack of adaptability also makes it impossible for the chip to protect itself against newer side channel attacks, becoming more susceptible to undesired physical attacks. In this concept of trust, a breach in the TPM can be extremely serious, since the computation is based on a blind concept of trust. For this reason, and depending on the environment the TPM is being used, it is important to have adequate countermeasures against side channel attacks.

The machine owner is obligated to use the Trusted Platform Module as a black box, having no knowledge on how the module is implemented and if it is properly implemented. Moreover, in the computational model proposed by the TCG, and when Remote Attestation is enforced, the users may have to relinquish control of the software applications that they can use. The users cannot modify the software being used, since that would invalidate the specific digital signature. This can be used, for example, to force the user to make undesired updates or even to force the use of a given application. An example of an unwanted use of this strict feature can be for censorship. Quoting Cambridge cryptographer Professor Ross Anderson "*someone who writes a paper that a court decides is defamatory can be compelled to censor it - and the software* *company that wrote the word processor could be ordered to do the deletion if she refuses*" [13].

The use of the Sealed Storage feature combined with the lack of control from the user, allows for an authorized user not to have access to is own data. An application can store a given data encrypted and locked to that application in that machine. So if an authorized user wishes to open these data with a different application or in a different machine, he will not be able to do so.

Another misuse of these features, from the user point of view, is the conditional access to a file, for example media file, that can only be read on a given machine and only by an application with Digital Right Management (DRM). An user cannot even copy the data from the digital I/O, since the Secure I/O is also out of his control.

1.1.2 Secure Computing on Reconfigurable Devices

Some of the drawbacks identified on the Trusted Computing Module can be overcame with the use of reconfigurable systems. Current reconfigurable systems are capable of achieving a computational performance, that allows them to replace dedicated hardware structures. With the implementation of the Secure Computing Module on a reconfigurable device, in particular in an FPGA, some of the drawbacks of an ASIC implemented chip can be significantly mitigated. The major advantage of such a reconfigurable approach is the fact that new algorithms can be easily and rapidly added to the architecture, thus allowing the module to be always updated. In addition to this, more efficient and secure computational structures can be added, as they are developed. If the module is to be used in an unsafe environment, where probing and physical attacks are possible, the designer/user can opt for structures with side channel resistant capabilities, in accordance to his requirements. Furthermore, the designer/user has the knowledge of the module internal structure, not seeing it as a black box. This can be very useful in avoiding the exploration of unknown backdoors to the system.

The adaptability and customization of such reconfigurable architecture, also allows the designer/user to select which features of the Trusted Computing are to be used or enforced. For example a bank may wish to run his own proprietary algorithms, while enforcing the use of the Sealed Storage in their mainframes.

Secure Computing

In this thesis the concept of Secure Computing (SC) is also introduced and used rather than Trusted Computer. Secure Computing is identical to Trusted Computing with owners override, differing in the fact that users have the possibility of having full control over their own machine.

Given that in Secure Computing the user can have full control over his own machine, the concept of blind trust on a remote computational system ceases to exist. Remote trustworthiness is no longer guaranteed.

The features of trusted computing presented in the previous section can now be under the user control and consequently under his decision. Features like Secure I/O and Sealed Storage are now used by option of the user, e.g. an user may choose to use Secure I/O for a wireless keyboard and not for viewing a movie. In Trusted Computing the features Secure I/O and Sealed Storage can be forced by choosing an external entity, for example due to Digital Right Management imposition.

The following describes the main proposed modifications on the Trusted Computing features to be used as Secure Computing.

Remote and internal attestation on SC: In Secure Computing two types of attestation are considered, *internal attestation* and *virtual remote attestation*. The internal attestation is used by the user to validate a given application. If the user decides to modify an application or use an alternative version of a given application, he still wants to be assured that the new application has not been tampered with by a third party. To guaranty this security, an internal Digital Signature for that (alternative) application is generated and safely stored. For this given application to be internally attested, the user simply has to compare the current Digital Signature with the internal Digital Signature of that application initially stored.

With the virtual remote attestation the user can make his chosen application ‘look like’ another application. This virtual Digital Signature is the signature seen by the remote machine, while the internal Digital Signature is used by the user to assure the integrity of his chosen application.

A modified version of an application can be provided and safely used, by generating an internal DS to validate this modified version and a Virtual DS (copy of the DS of the original application), used to authenticate the application to an external challenger. An example of this attestation schema would be the use a web browser modified to have different features but still valid to access a web banking system. The user can now decide if the application he is using has the same security as the application requested by the remote challenger. In the internal attestation a simplified DS can be used, since the DS values can be locally and securely stored.

Endorsement key on SC: In TC the Endorsement key is used to identify the TPM; each TPM chip has its own unchangeable key. In SC the user can have the capability to read and modify this key, thus having the possibility of creating as many virtual Secure Computing Modules has needed. With this, privacy issues are no longer a concern, since to each remote challenger the user can identify himself as a different user/machine.

Sealed storage: The key used to encrypt the data can be accessed by an authenticated user. The user has full control over the data, meaning that the user can export or import data to and from any application or machine. The conditional access to a given set of data is no longer under the full control of a given application; an authenticated user has full access over all his own data.

Secure I/O: Identically to the sealed storage, in SC the user has control over his own data. The user has control over the encryption of the transmitted data. It is the user's decision, whether or not to cipher the data to be transmitted, not the application.

Drawbacks of SC on reconfigurable devices

The implementation of the Secure Computing on reconfigurable technologies also presents some disadvantages and technical problems. One of the most relevant ones is the volatility of the memory in reconfigurable devices, making it necessary to upload the initial configuration onto the device every time the device is switched on.

Configuration files are needed and time required to upload these configurations. The several configuration files for each configurable structure have to be stored somewhere in order for them to be uploaded into the device. Additionally, the computation of a given hardware implemented function can only start after the structure has been properly loaded into the device. This configuration time affects the performance of the whole system.

Given that this computational platform may be subjected to malicious attacks, the partial configurations loaded into the reconfigurable device have to be attested in order to assure their correct functionality. Due to these possible attacks, the initial configuration of the reconfigurable device has to be properly stored and loaded to the device in a secure fashion.

Even though typical reconfigurable devices such as Field Programmable Gate Array (FPGA) have an area ratio of about 10 times that of an equivalent Application Specific Integrated Circuit (ASIC) chip, their reconfiguration capabilities allow to only load the configuration relative to the part of the circuit that is required. As a consequence, the implementation of a SCM in an FPGA can actually reduce the total area required to implement the SCM, since only a fraction of the circuit is required to be loaded at a given time.

Using the Secure Computing concept, rather than the Trusted Computing one, Remote Attestation can no longer be guaranteed, causing a loss of trustworthiness in the remote computation. For example, the full trustworthiness in remote public computers no longer exists.

1.2 Secure Computing Module

The proposed approach is user oriented, which means that the user has the control over the system unlike the strictly closed system of the TCM. This thesis is focused on the functionality and the adaptability of the system, also allowing new algorithms and standards, while maintaining a strong backward compatibility. The security of the proposed computing module is mainly supported by cryptographic algorithms, providing a computational organization that allows these algorithms to be properly used, while assuring the SCM's reliability and performance.

Some functionalities of the TCM are disregarded in the first version of the proposed SCM. The most significant one is the memory curtaining functionality; the memory curtaining can be implemented in software without significant performance degradation. Additionally, some General Purpose Processors (GPP) already have similar mechanisms in their core architectures. The concept of memory curtaining is not algorithm dependant, thus, no significant changes are expected in future versions of this module. This means that it is best to implement this mechanism in the static part of the processing module.

1.2.1 Architecture Overview

Figure 1.1 depicts the conceptual organization of the proposed Secure Computing processing module. This structure is composed by: (i) a General Purpose Processor (GPP); (ii) an internal memory to store critical values, part of which non-volatile; (iii) reconfigurable Cryptographic Computational Units

(CrCUs); (iv) a control unit; (v) a self attestation block; (vi) an I/O Interface for dedicated peripheral devices; (vii) a main data memory.

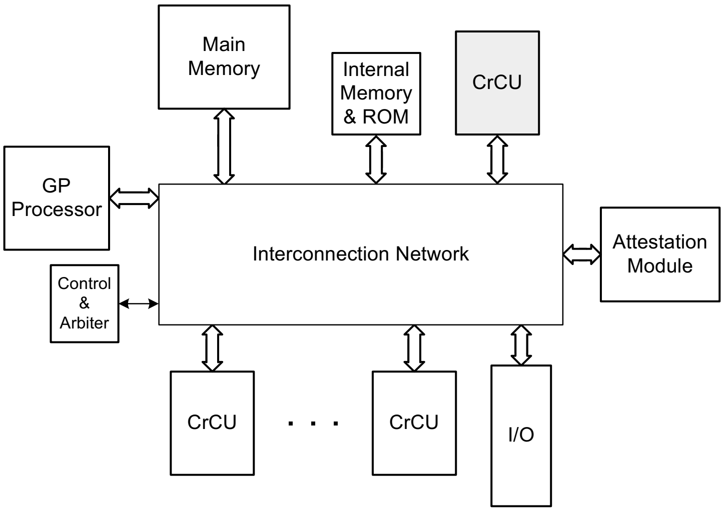


Figure 1.1: Secure Computing Module organization.

The GPP in this structure is the one responsible for the execution of non critical parts of the execution path, as well as for the data flow and for choosing the algorithms to be used. Non critical computation includes key management, key generation and other parts of high level security protocols. The non critical parts of the execution path are executed in software. The critical parts are executed in hardware, in this particular case the cryptographic functions. In the proposed SCM, the interchanging between hardware and software is supported by the use of the Molen paradigm to computation [14].

1.2.2 Attestation Mechanism

The CrCUs are the ones responsible for the execution of the cryptographic algorithms, which can be symmetrical or asymmetrical algorithms and hash functions. These units are configured according to the protocol being used and the required algorithms. For example, one application might use a protocol that requires the use of 3DES/RSA1024/SHA-128, in which a 3DES encryption core and a SHA128 hashing core are loaded to the CrCUs, while another one might require AES/RSA1024/Whirlpool, in which case the 3DES and SHA128 cores are replaced by AES and Whirlpool cores.

One of the features of SC is the attestation of the functions that are being executed. In a hybrid HW/SW computing approach, an application can be executed either in software or in hardware. Thus, the test of an application consists not only in the validation of the software being executed but also in the validation of hardware being used. At the software level the attestation is performed by generating a Digital Signature (DS) of the execution path being executed, and comparing it with the expected value. For the hardware part of the application or algorithm the same approach can be taken, taking into account that a reconfigurable structure/technology is being used. This hardware attestation can be performed by generating the DS of the configuration bitstream of the respective CrCU, whenever this is uploaded to the reconfigurable logic, followed by an identical test as the one performed for the software.

In order to assure the security of the SCM, a predefined skeleton structure of the SCM has to be properly loaded into the device. Once it is loaded, this structure is responsible for the validation of the next configuration to be loaded and for the assurance that it is an untampered one. Every new configuration is tested by the current existing structure, thus assuring that a new loaded configuration is an untampered one. Since it is not efficient to have the DS of every core or code created with every hashing algorithm, it becomes necessary to have a permanent CrCU to perform the computation, represented by the grey CrCU in Figure 1.1. This maintains the independence of the attestation unit and the flexibility of the SCM.

1.2.3 Memory and I/O Organization

In the proposed organization two classes of memory exist. The external memory (exRAM), with a large capacity, that does not need to be within the secure environment of the internal chip, and the internal memory. The internal memory is composed of two different types of storage, volatile internal RAM (iRAM) and non-volatile memory (ROM).

The internal RAM is used to improve the implementation of the sealed storage feature. Whenever the data is to be stored encrypted in memory, the internal memory can be used has a temporary buffer. Thus, the computational units access the decrypted data from this memory, that is only accessible from within the system. The reading and writing to the main memory is performed by an encryption/decryption CrCU. In this way, the data is only decrypted within the internal security of system. Identically, this internal memory is also used in the secure I/O feature, along with the CrCU units and the I/O hardware of the system. The I/O data is not directly read to the unsecured external memory.

The ROM is used to store the root DS values and critical user passwords. The initialization organization is also stored in this internal ROM. Most of current reconfigurable chips do not include non-volatile memory components; this makes the system unsafe to external physical attacks. However, the industry is starting to produce FPGA chips with internal non-volatile memories that allow the storage of configurations bitstreams and other data. In these devices the problem of the initialization and storage of critical data is solved, taking into account that the internal data can be made unreadable after it has been written.

1.3 Dissertation objectives

The work proposed in this Doctoral thesis targets the design of a Secure Computing (SC) hardware structure for reconfigurable devices.

The proposed Secure Computing Module allows the concept of Secure Computing to be used on a large variety of reconfigurable computational systems, such as soft-cores, polymorphic systems [14], as well as in non reconfigurable systems as a customizable replacement of the TPM chip. The reconfiguration allows the system to be easily updated whenever a new version of a protocol or cryptographic algorithm is specified. Specific systems that require some of the features of the trusted computing module can efficiently use the proposed Secure Computing structure, by selecting and using only the features required for their systems.

The research efforts concerning the implementation of cryptographic algorithms have been mostly focused on the design of efficient symmetrical algorithms and hashing functions, since Secure Computing is mostly supported by them. The other major goal of this thesis is the design of an efficient mechanism for the attestation of the dynamically reconfigurable hardware, in order to assure a reliable operation of the Secure Computing Module. The following describes the main objectives of this dissertation.

Design of a Secure Computing Module: The core objective of this dissertation is the design of a computational module capable of facilitating the usage of the Secure Computing features in a reliable fashion. A key feature of this module is the capability of adapting its structure and computational characteristics to a wide range of applications and systems. The implementation of the proposed structure on reconfigurable systems gives a considerable versatility to the proposed SCM; however, with this arises the need to attest the dynamically loaded hardware, not just the executed software. A prototype of the proposed Secure Computing Module is developed for the Virtex II Pro FPGA technology from Xilinx [15]. Experimental results suggest that with a middle range device (XC2VP30-7) in this technology, the proposed features can be implemented and high performance can be achieved.

Investigate ciphering algorithms: Investigate which ciphering algorithms are required to implement the Secure Computing features. The three major classes of ciphering algorithms, private key algorithms, public key algorithms, and hash functions, are studied in order to devise which are better suited to implement the Secure Computing features.

Design performance efficient ciphering structures: In order to allow the Secure Computing features to be used without significantly affecting the performance of the system, high performance ciphering cores must be designed. Part of the research work developed in this thesis is focused on the design of performance and area efficient ciphering structures. These structures are capable of achieving high throughputs at low area cost, on reconfigurable devices.

Devise a hardware attestation mechanism: In the context of this thesis, all data outside the internal security of the chip cannot be considered safe. In order to assure that the data, including the data describing the execution path, have not been tampered with, they must be authenticated before they are used. While for software code several methods have been proposed [16, 17], the authentication of hardware is still not used. In this dissertation, a hardware attestation module is proposed. The proposed structure allows the attestation of dynamic reconfigurable hardware structures in a semi transparent manner and with negligible performance degradation. With this mechanism the internal attestation feature of Secure Computing is assured.

1.4 Dissertation overview

This thesis is organized as follows. The next chapter presents an overview of the cryptographic algorithms and their usage.

Chapter 3 presents the cryptography cores developed for the most usual symmetrical algorithms, namely DES and AES.

Chapter 4 describes the cryptography cores developed for the most frequently used hashing functions, as well as, an implementation of one of the hashing functions that will most likely become part of future standards, the Whirlpool hashing function.

In Chapter 5, an micro-architectural mechanism is proposed for the self attestation of the Cryptographic computational units of the proposed processor.

Finally, in Chapter 6, the proposed Secure Computing hardware structure is described and evaluated with the designed computational units and the proposed attestation mechanism.

Chapter 7 concludes this thesis with a summary of the obtained results, the contributions of this thesis and some future research directions.

Chapter 2

Cryptographic Algorithms

Contents

Cryptographic systems support numerous security critical applications, ranging from highly secure uses, such as banking transactions, to low security applications such as television broadcast. Cryptographic algorithms can be divided into three major classes: private key algorithms, public key algorithms, and hash functions [18, 19].

The private key algorithms, also known as symmetrical encryption algorithms, require the same key for the encryption and for the decryption. This means that the key must be kept private, known only by the users that have to encrypt or decrypt the message. Even though computationally demanding, these are the algorithms commonly used in the encryption of the main data streams.

The public key algorithms use two sets of keys; one is used to encrypt while the other one is used to decrypt a given message. If a data block is encrypted with one of the keys, the other key has to be used in the decryption. This means that one of the key can be made public, facilitating the exchange of data over unsecured communication channels. This type of algorithms tend to be computationally heavy, thus only small amounts of data are typically encrypted with this type of algorithm.

While the first two types of algorithms are used to encrypt and decrypt data, the hash functions are one-way functions that do not allow the processed data to be retrieved. Hash functions have the particularity of generating a small fixed length output value, the footprint or digest message, that is highly correlated with the input data. The most important characteristics of these functions is the fact that virtually no information about the input data can be obtained from the outputted hash value. An adequate hash function has a very low probability of two different input data streams generating the same hash value.

This Chapter is divided into three major sections, one for each class of ciphering algorithms.

2.1 Symmetrical Algorithms

Symmetric encryption algorithms are the algorithms used to cipher the bulk of data that has to be sent through an insecure channel. In these algorithms, only one key exists, a single private key used both for encryption and decryption. A variety of symmetric encryption algorithms exist such as AES, Blowfish, RC4, DES, and IDEA. The most commonly used are the Data Encryption Standard (DES), its variation 3DES, and its new replacement the Advanced Encryption Standard (AES) [19].

Nowadays, the field of cryptography is growing up very intensively and many others algorithms have been presented to meet the requirements of modern electronic systems. The high performance cryptographic architectures are based on ASIC and FPGA technologies. In both cases, for each new solution is necessary to keep the compatibility with devices which are already available on the market. In this thesis, implementations of the DES and AES algorithm, as part of a secure computing system based on FPGA technology, are presented. In this section the different encryption modes used in symmetrical ciphering are presented and discussed.

2.1.1 DES

Since the DES algorithm was introduced in 1976, many devices and systems have based their security in this algorithm. In 1998 [20] the DES algorithm and the 54 bit encryption key, was deemed unsafe and replaced by 3DES, which essentially consists in performing the DES computation three times with 2 or 3 different keys, which correspond to a 112 to 168-bit equivalent key. With the increase of embedded application requiring DES (and 3DES), like RFID and bank cards, efficient hardware implementations of DES are required. The DES computational structure has the advantage that encryption and decryption operations are very similar, requiring only the reversal of the key schedule, operation that does not require any computation.

DES Encryption

In DES, 64 bit data blocks are encrypted using a 54 bit Key (obtained from an input key with 64 bits). The intermediate ciphered values are processed as two 32-bit words (L_i and R_i), which are manipulated in 16 identical rounds as depicted in Figure 2.1. This manipulation consists of substitution, permutation, and bitwise XOR operation, over the 64-bit data block. The DES algorithm also has an Initial bit Permutation (IP) at the beginning of a block ciphering. To conclude the ciphering of a block, a final permutation is performed, which corresponds to the inverse of the initial permutation (IP^{-1}). A higher data scrambling is achieved with these permutations. An interesting characteristic of the initial permutation is that the odd and even bits of the 64 bit data block are split and placed either on the right or on the left 32 bit path. The main computation performed in each of the 16 rounds is designated by Feistel network, named after cryptographer Horst Feistel. In the pseudo-code of Figure 2.2, describing the DES algorithm, the Feistel network computation is performed

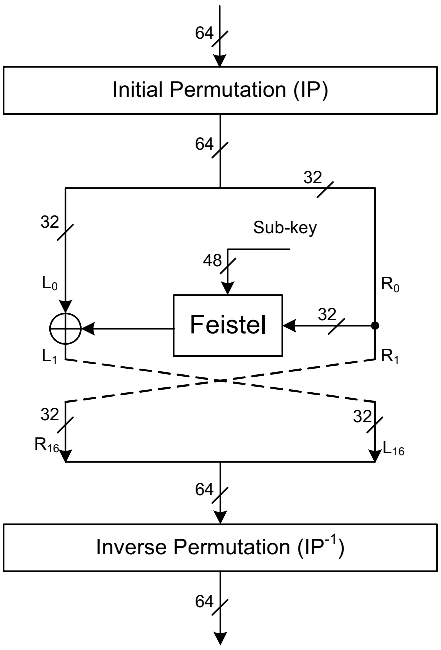


Figure 2.1: DES computation.

by the function $f(R_{i-1}, K_i)$.

```
$L_0 \& R_0 = IP(input)$
for i=1 to 16 do
    $L_i = R_i$
    $R_i = L_{i-1} \oplus f(R_{i-1}, K_i)$
end for
output = $IP^{-1}(L_{16} \& R_{16})$
```

Figure 2.2: DES Pseudo-code.

In each round a different sub-key (K_i) is used, generated from the main key expansion. The round main computation or Feistel network is depicted in Figure 2.3. The Feistel network is composed by the 3 main operation in symmetrical ciphering, namely key addition, confusion, and diffusion [21]. The first half of the round block (R_i) is expanded from 32 to 48 bits and added to the 48-bits of the current sub-key. While the data expansion can be hardwire implemented, XOR gates have to be used for the key addition. The Key addition

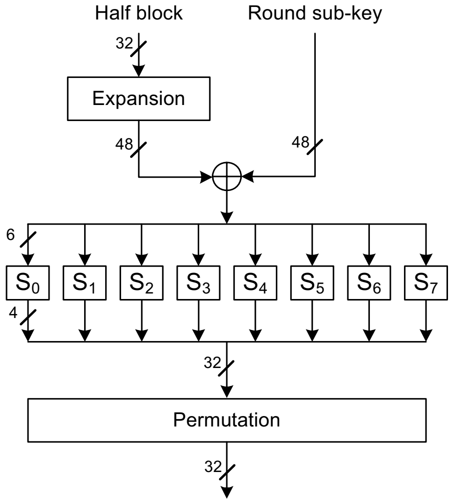


Figure 2.3: DES Feistel network.

operation is followed by the confusion operation, performed by Substitution BOXes (SBOXs). In this operation the value resulting from the addition is grouped in 8 blocks of 6 bits each. Each group of 6 bits is replaced by a different set of 4 bits, resulting in 32 bits. The diffusion operation is performed by a final permutation in the Feistel Network.

The following describes in more detail each of the operations performed in the DES Feistel network.

Expansion: The expansion of the 32-bit input word R_i to a 48-bit word is accomplished by permutating and replicating some of the input bits, as described in Table 2.1

Table 2.1: DES expansion regime.

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

Substitution: The substitution is performed by the S-Boxes S_1, ..., S_8. Each S-Box is a fixed 4×16 table, addressed from the 6-bit input word. The S-box computation can be performed by lookup tables, with 4 rows and 16 columns. Each S-Box has its own configuration [22]. The 4-bit output of these 8 S-boxes form the 32-bit word used in the permutation function.

Permutation: In the permutation operation, the input bits are permuted. Each input bit is placed in a different position, according to the values presented in Table 2.2. Note that this is a plane bit permutation, without any bit replication.

Table 2.2: DES Feistel network permutation regime.

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

DES Key Schedule

From the 64 bits received for the input key, only 58 bits are used in the encryption process: the remaining 8 bits are parity-check bits use for error detection. The parity bits are disregarded in the computation of the key schedule. After discarding the parity bits, the remaining 56 bits (K) are used to generate the key schedule composed by 16 sub-keys.

In order to generate these sub-keys (K_i), several steps are necessary. First the PC-1 permutation, given in Table 2.3, is applied to the input key. The result can be described as $C_0 D_0 = PC-1(K)$, were C_0 contains the first 28 bits of $PC-1(K)$ and D_0 the last 28 bits. Afterwards, it is necessary to perform a Left Shift (LS) of 1 or 2 positions to each of the C_i and D_i. Finally, the 48-bit sub-key k_i is given by the permutation PC-2, presented in Table 2.4. The key expansion computation is described in Figure 2.4

Table 2.3: DES PC-1 permutation.

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

Table 2.4: DES PC-2 permutation.

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

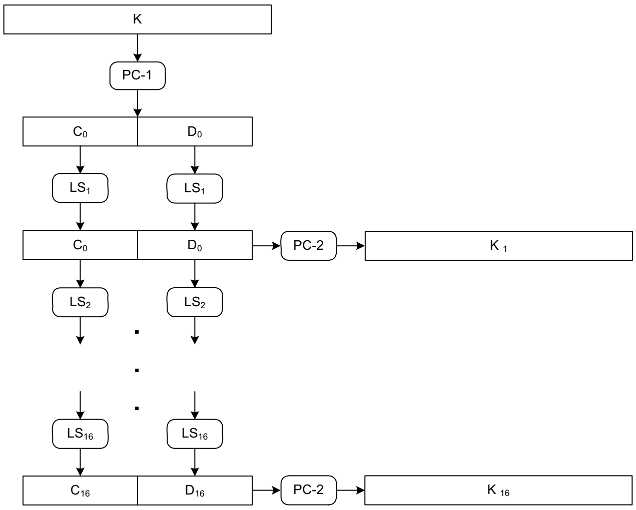


Figure 2.4: DES sub-key generation

DES decryption

In order to obtain the initial data from the ciphered data stream, the same algorithm is used. It only differs in the order in which the expanded keys are used.

In this case, the keys are used in the inverse order, regarding the encryption calculation. In conclusion, for the decryption the data goes through the same computational data path as the data being encrypted (see Figure 2.1), but using the expanded key in the inverse order.

Triple-DES

Triple-DES (or 3DES) has been developed to cope with the relatively small key size of the original DES algorithm, considering present computational power, while maintaining compatibility with the original DES algorithm and the existing systems for implementing it.

3DES encrypts data applying the standard DES encryption three times, over the same data block with at least two different keys. With this method, 3DES can maintain the compatibility with DES while increasing the key security. In 3DES, keys of 112 or 168 usable bits are employed.

2.1.2 AES

The AES is the new NIST standard chosen to replace DES [22]. It uses the Rijndael encryption algorithm with cryptography keys of 128, 192, or 256 bits; the 128 bit key is the most commonly used key size. As in most of the symmetrical encryption algorithms, the AES algorithm manipulates the 128 bits of the input data, disposed in a 4 by 4 bytes matrix, with byte substitution, bit permutation and arithmetic operations in finite fields, more specifically, addition and multiplications in the Galois Field 2^8 ($GF(2^8)$). Each set of operations is designated by a round. A round computation is repeated 10, 12 or 14 times depending on the size of the key; 128, 192, or 256 bits respectively.

AES Encryption

The coding process includes the manipulation of a 128-bit data block through a series of logical and arithmetic operations. The encryption pseudo-code is depicted in Figure 2.5. The following describes in detail the operation performed by the AES encryption in each round. The State variable contains the 128-bit data block to be encrypted.

SubBytes transformation: The replacement of one set of bits by another is a non linear transformation. In the Rijndael algorithm, this replacement is performed over a set of 8 bits. This replacement can be described by an affine

```
State = in

AddRoundKey(State, key[0 $to$ Nb−1])
for round= 1, round<Nr, round=round+1 do
  SubBytes(State)
  ShiftRows(State)
  MixColumns(State)
  AddRoundKey(State,key[round×Nb $to$ (round+1)×Nb−1])
end for

SubBytes(State)
ShiftRows(State)
AddRoundKey(State,key[Nr×Nb $to$ (Nr+1)×Nb−1])

out = State
```

Figure 2.5: Pseudo Code for AES Encryption.

transformation (over $GF(2)$), as presented in (2.1):

$$\begin{aligned} b_i^{'} = b_i \oplus b_{(i+4)mod_8} \oplus b_{(i+5)mod_8} \oplus b_{(i+6)mod_8} \\ \oplus b_{(i+7)mod_8} \oplus c_i \; ; \; 0 \leq i < 8, \end{aligned} \tag{2.1}$$

where b_i is the i-th bit of byte b(x), obtained from the State array. This byte substitution is performed over each byte individually. The c_i is the i-th bit of the constant value $\{01100011\}$. The byte substitution operation can be implemented in hardware by a 256 byte lookup table, with an 8 bit input and an 8 bit output, or by using logic operators (2.1).

ShiftRows: The bytes in each row of the state matrix are rotated to the left by $0, 1, 2$ or 3 byte positions, depending on the row where they are located, as depicted in Figure 2.6. For example $S_{1,0}S_{1,1}S_{1,2}S_{1,3}$ is transformed to $S_{1,1}S_{1,2}S_{1,3}S_{1,0}$. Since this operation contains no calculations, it can be implemented simply by routing the appropriate byte from the output of the previously described lockup table to the corresponding input of the MixColumns unit.

MixColumns transformation: In this transformation, each column is treated as a four-term polynomial over $GF(2^8)$ and multiplied modulo $x^4 + 1$ with a fixed polynomial $a(x)$, given by:

$$a(x) = 03x^3 + 01x^2 + 01x + 02 \tag{2.2}$$

The resulting new column is thus calculated with the previous values of only that column. The calculation of the new column, presented in (2.3), is performed over the GF(2^8), where the multiplications ($\bullet$) are performed by AND operations and the additions and subtractions ($\oplus$) by XOR operations.

$$\begin{aligned}
S'_{0,c} &= (02 \bullet S_{0,c}) \oplus (03 \bullet S_{1,c}) \oplus S_{2,c} \oplus S_{3,c} \\
S'_{1,c} &= S_{0,c} \oplus (02 \bullet S_{1,c}) \oplus (03 \bullet S_{2,c}) \oplus S_{3,c} \\
S'_{2,c} &= S_{0,c} \oplus S_{1,c} \oplus (02 \bullet S_{2,c}) \oplus (03 \bullet S_{3,c}) \\
S'_{3,c} &= (03 \bullet S_{0,c}) \oplus S_{1,c} \oplus S_{2,c} \oplus (02 \bullet S_{3,c}) \qquad (2.3)
\end{aligned}$$

Since the multiplication of two bytes results in a double byte number, the result is replaced by the remainder polynomial, that in the case for the Rijndael algorithm is given by the irreducible polynomial:

$$m(x) = x^8 + x^4 + x^3 + x + 1. \qquad (2.4)$$

This calculation can be performed by subtracting the $m(x)$ polynomial whenever the result of each partial multiplication is bigger than FF. Finally, the addition of the four coefficients of the polynomial can be directly performed by using XOR gates.

AddRoundKey: The final operation to be performed in each round is the addition (XOR in $GF(2^8)$) of the respective round Key to each column of the State matrix. Each round Key consists of four 32-bit words from the expanded Key (xK). The formalized operation is:

$$\begin{aligned}
[S'_{0,c}, S'_{1,c}, S'_{2,c}, S'_{3,c}] =& [S_{0,c}, S_{1,c}, S_{2,c}, S_{3,c}] \oplus \qquad (2.5) \\
& [xK_{round \times N_b + c}] \;\; ; \;\; 0 \leq c < N_b
\end{aligned}$$

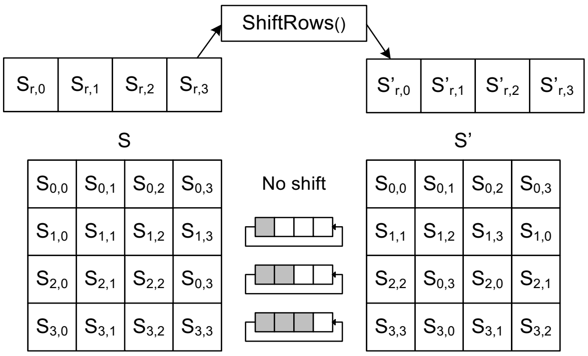


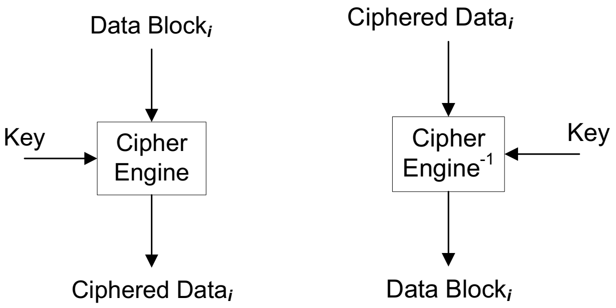
Figure 2.6: AES ShiftRows.

AES decryption

The decryption process is identical to that of the encryption (see the pseudo code in Figure 2.5). The main differences in the decryption computation lay on the byte substitution and on the polynomial equation used in the column mix. The byte substitution transformation for the decryption has the same structure as the encryption, only differing in the values of the look up table, presented in [23]. The row shifting is also identical, with the only difference that, the rotation of the byte is performed to the right and not to the left, as depicted in Figure 2.6. In the inverse column mix transformation, the computation is exactly the same as in the encryption, differing only in the polynomial coefficient values.

2.1.3 Encryption Modes

In symmetrical encryption, large blocks of data are ciphered in small blocks that typically vary from 64 bits (in DES) to 128 (for AES). Since the encryption of a data stream is performed with the same private key, some information about the data can leak. An input block with a given binary sequence will always result in an output block with a fixed sequence, if nothing is done to prevent it. This describes the most basic encryption mode designated by Electronic CodeBook (ECB). In the ECB mode, a data stream is split into smaller data blocks. Each of these smaller data block is individually feed to the encryption function and ciphered. The ciphered block i has no correlation or dependency over any other data block. The encryption/decryption operation for this mode is depicted in Figure 2.7.

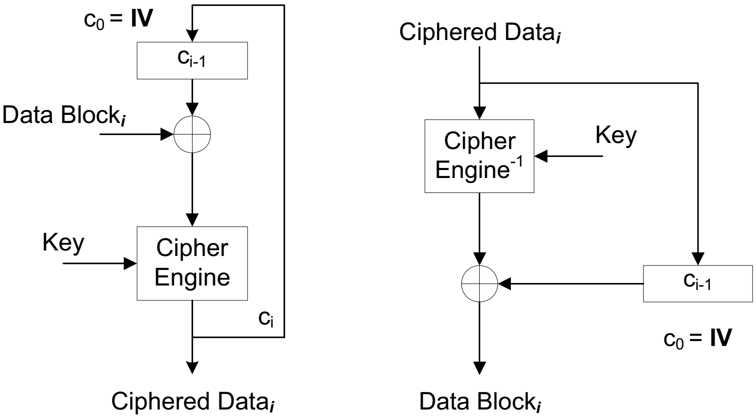


(a) Encryption (b) Decryption

Figure 2.7: ECB mode.

To avoid the leakage of patterns from the output data stream, several more complex cipher modes have been developed. In these modes, an output data block i, depends not just on the input data block, but also on the previous data blocks and the internal value. In all these modes, an initialization value, designated by Initialization Vector (IV), is required. If the same IV is used for different data streams some leakage may still occur.

One of the earliest and frequently used modes is the Cipher-Block Chaining (CBC) mode. In this mode, the current data block i is bit-wise XORed with the previous encrypted data block; for the first data block the IV is used instead of the previous data block, since non exists for $i < 1$. This operation is better illustrated in Figure 2.8. When decrypting a message, the inverse operation is performed; after decrypting data block i the values is XORed with the previous $i-1$ encrypted data block.



(a) Encryption (b) Decryption

Figure 2.8: CBC mode.

The following describes other block cipher modes with data feedback. In Cipher FeedBack (CFB) the data block to be encrypted is XORed after the encryption computation, not at the begging as is the case in the CBC mode.

The Output FeedBack (OFB) mode is structurally identical to the CFB mode; however, the value propagated to the next data encryption operation is the output of the encryption engine, before the XORing with the input data block.

In the CounTeR (CTR) mode, the input of the encryption engine is given by successive increments of the IV; the remaining computation is identical to the CFB mode.

In this thesis only the two most common modes, ECB and CBC, are considered. In terms of hardware complexity and computational delay, due to an increased data path, the CBC depicts the worst case scenario. Only the CTR mode may have a slightly higher area requirement due to the implementation of a counter; no delay increase is expected.

2.2 Asymmetrical Algorithms

Asymmetric encryption algorithms are a class of encryption algorithms that have the particularity of having two distinct key sets. A private key that has to be maintained secret and a public key published to any one who wishes to send an encrypted message. If a message is encrypted with one of the key pairs, the other key pair must be used in the decryption process. These algorithms use very large keys and require a significant amount of computational power, when compared with the symmetric encryption algorithms. Typically, asymmetrical algorithms are used to safely transfer small amounts of data over unsecured channels, e.g. the digital signature of a data stream or a private key for symmetrical encryption algorithms. Due to their computational requirements, asymmetrical algorithms tend to be quite inefficient when ciphering large data blocks.

2.2.1 RSA

The RSA [18] cryptographic system is still one of the most used public-key ciphers. RSA security is based on the difficulty in the factorization of large numbers; usually 515, 1024, or 2048-bit keys are used. Current factoring algorithms are able to factor number up to 130 decimal digits in contrast to the 154 digits obtained from a 512-bit number. The ciphering process is based in modulo and exponentiation (i.e. multiplications) operations, and can be summarized by (2.6).

$$x = y^a \mathrm{mod}\, n = (x^b \mathrm{mod}\, n)^a mod\, n \tag{2.6}$$

RSA public and private Keys

In this system the Key generation and calculation is based on the generation and manipulation of 2 distinct odd prime numbers p and q. With these 2 numbers the values n and $\phi(n)$ are calculated as:

$$n = p \times q \tag{2.7}$$

$$\phi(n) = (p-1)(q-1). \tag{2.8}$$

The public Key is composed by the pair (n, b), were b is a random number with the following characteristics:

$$1 < b < \phi(n); \tag{2.9}$$

$$gdc(b, \phi(n)) = 1, \tag{2.10}$$

meaning that b has to be a positive number smaller than $\phi(n)$, and relatively prime to $\phi(n)$ (b and $\phi(n)$ cannot have any common divider). In order to verify that b and $\phi(n)$ are relatively co-prime numbers, the Euclidean algorithm can be used in order to verify that $gdc(\phi(n), b) = 1$. Note that b has to be an odd number, since $\phi(n)$ will always be even (see equation 2.8).

Finally the private Key is composed by the pair (n, a), were a is the multiplicative inverse of b given by:

$$(a \times b) \, mod \, \phi(n) = 1 \tag{2.11}$$

$$a = b^{-1} mod \, \phi(n), \tag{2.12}$$

thus $ab \equiv 1(mod \, \phi(n))$. (2.12) can be rewritten as:

$$a \times b - \phi(n)k = 1 \Leftrightarrow 1 = \phi(n)k + b \times a \quad ; \, -k, a \in N, \tag{2.13}$$

thus by using the Extended Euclidean algorithm, the value k and most importantly a (b^{-1}) can be calculated. Other methods can be used in the RSA calculations, such as the Chinese Reminder Theorem (CRT) and alternative multiplication techniques, to speed-up the computation of this algorithm. One of these alternative multiplication techniques or arithmetic systems used to improve the RSA performance and security is the Residue number system (RNS). This arithmetic system have the characteristics that large numbers are decomposed into smaller parcels and processed in parallel, without the need of carry propagations between each parcel. This makes the computation of the RSA faster [24, 25] and more area efficient, but also more secure to side channel attacks [26, 27]. Appendix A presents the work developed on RNS, in particular in optimizing the multiplication of large operands.

Encryption and Decryption

Data encryption and data decryption are performed with the exact same algorithm, differing only on the Key. This process is based on exponent and modulo calculations, in order to compute:

$$(data^{key}) mod\, n. \tag{2.14}$$

These two operations could be simply performed by a processor if the operand were small, however they are extremely wide, typically $data$ and n are a 200 digit number and key can have between 512 and 2048-bit number. In order to simplify these operations the following mathematical properties can be used:

$$b = \sum_{i=0}^{k} 2^i \times b_i; \tag{2.15}$$

$$x^b = \prod_{i=0}^{k} x^{b_i} = \prod_{i=0}^{k} x_i; \tag{2.16}$$

$$\left(\prod_{i=0}^{k} x_i\right) \bmod n = \prod_{i=0}^{k} (x_i \text{mod } n)\,. \tag{2.17}$$

This allows the use of square-and-multiply algorithm, depicted in Figure 2.9, thus reducing the number of modular multiplication from b–1 to $2k$, which can be quite significant, since $b \simeq n$ and $k \simeq \lfloor log_2(n) \rfloor + 1$. With this algorithm it is possible to compute $data^{key} mod\, n$ in $\mathcal{O}(k^3)$

```
$y = 1$
for $i = 0$ to $k - 1$ do
    $y = y^2\ mod\ n$
    if $b_i = 1$ then
        $y = (y \times x)\ mod\ n$
    end if
end for
```

Figure 2.9: Square-and-multiply algorithm for $y = x^b mod\, n$.

The data being ciphered has to be partitioned into smaller data blocks; the dimension of each data block has to be smaller than $\phi(n)$.

RSA example

In order to better illustrate this ciphering algorithm a small example is presented. Suppose that a Receiver wishes to create a pair of RSA keys. He can chose a random $p = 101$ and $q = 113$, thus obtaining $n = 11413$ and $\phi(n) = (p - 1) \times (q - 1) = 100 \times 112 = 11200$. In order to have the key pairs complete (public and private keys), a value for b has to be chosen and its corresponding inverse. Randomly choosing $b = 3533$ $(1 < 3533 < \phi(n))$ and verifying that $gcd(\phi(n), b) = 1$ the multiplicative inverse is calculated by the Extended Euclidean algorithm, that yields $a = b^{-1} = 6597$. The receiver now has a private key $(a, n) = (6597, 11413)$ that will not be shared and the public key $(b, n) = (3533, 11413)$ that can be shared with the Sender via a possibly unsecured channel. Suppose a Sender wishes to send a message (the plain text *9726*), he simply has to calculate the cipher text with the public key $(3533, 11413)$ by computing:

$$9726^{3533} \, mod \, 11413 = 5761,$$

the cipher text (5761) is safely sent through the non secure channel and received by the Receiver, who will decrypt the message using is private key (6597,11413):

$$5761^{6597} \, mod \, 11413 = 9726,$$

thus safely obtaining the message (the plain text 9726) sent by the Sender.

2.2.2 ElGamal

The ElGamal cryptosystem [28] is based on the discrete logarithm problem, that consists in the difficulty of calculating the discrete logarithm of numbers sufficiently large (at least 150 digits), while the inverse operation of exponentiation can be computed efficiently by the square-and-multiply method. The ElGamal cryptosystem is non-deterministic, since the cipher text depends on both the plain text and on a random value (k chosen by the Sender). A disadvantage of the ElGamal system is that the encrypted message becomes very big, about twice the size of the original message m.

ElGamal public and private Key

In this encryption algorithm the public key is composed by 3 numbers and the private key by 2. Unlike the RSA algorithm, this algorithm only requires the generation of one prime number. This number has to be sufficiently wide, in order to make the resolution of the discrete log problem computationally unfeasible. The private key is also composed by the value a, a random positive integer smaller than $p-1$ $(0 \leq a \leq p-2)$. Thus the private key is given by the pair (p, a). Note that the value a has to be kept private.

The public key is composed by three values, the value p already determined, the value α and the value β. $\alpha \in Z_p^*$ is a primitive element, and can be chosen randomly. Finally the value β has to be calculated by:

$$\beta = \alpha^a \, mod \, p \ , \tag{2.18}$$

resulting in the public key (p, α, β) is obtained.

Encryption and Decryption

The calculations performed in the ElGamal cipher are identical to those of the RSA cipher; however, the algorithm itself is different. The encryption and the decryption algorithms are different.

The encryption of a plain text is divided in two operations, resulting in a cipher text divided in two messages. In order to compute the cipher text, a random number k has to be generated with $0 \leq k \leq p-1$. The number k is secret, only known by the Sender. This value is discarded after each cipher text block, thus a new random value for k has to be generated for each data stream. The first part of the cipher message is y_1, given by:

$$y_1 = \alpha^k \, mod \, p. \tag{2.19}$$

y_1 is calculated, identically to the RSA algorithm, by the square-and-multiply algorithm. The second part of the cipher message is y_2, given by:

$$y_2 = x\beta^k \, mod \, p, \tag{2.20}$$

that can be written as:

$$y_2 = (x(\beta^k \, mod \, p)) \, mod \, p = xK \, mod \, p. \tag{2.21}$$

To obtain this value, the square-and-multiply algorithm can be used, in order to calculate $K = \beta^k \, mod \, p$, followed by a multiplication modulo p. Note that the ciphered message is twice as long as the RSA equivalent, also doubling the required computation.

The decryption algorithm manipulates both messages of the cipher text, along with the private key (p, a), in order to obtain the original plain text (x). The recuperation of the plain text is given by:

$$x = y_2(y_1^a)^{-1}\, mod\, p. \tag{2.22}$$

This can be written as:

$$x = y_2K^{-1}\, mod\, p = (y_2(K^{-1})\, mod\, p)\, mod\, p, \tag{2.23}$$

where:

$$K = y_1^a\, mod\, p = \beta^k\, mod\, p \tag{2.24}$$

Once more, this calculation can be performed with the square-and-multiply algorithm. To calculate the multiplicative inverse of K, the Extended Euclidean algorithm can be used.

ElGamal example

To following illustrates this ciphering calculation. Suppose that a Receiver wishes to create a pair of keys. He chooses $p = 97$, $\alpha = 5$, and a random $a = 58$, thus obtaining $\beta = 5^{58}\, mod\, 97 = 44$ through the Extended Euclidean algorithm. He publishes the public key (p, α, β)=$(97, 5, 44)$, and keeps the private key (p, a)=$(97, 58)$.

The Sender how wishes to send the Receiver the plain text $x = 3$, reads the Receiver public key through a public channel. The Sender chooses a random value $k = 44$ and computes:

$$y_1 = 5^{36}\, mod\, 97 = 50$$

and

$$y_2 = 3 \times (44^{36}\, mod\, 97)\, mod\, 97 = 3 \times 75\, mod\, 97 = 31.$$

These two values $(50, 31)$ are sent to the Receiver through a public channel.

The Receiver receives these two values and computes:

$$K = 50^{58}\, mod\, 97 = 75 \Rightarrow K^{-1} = 22\, mod\, 97$$

and finally calculates the plain text:

$$x = 31 \times (50^{58})^{-1}\, mod\, 97 = 31 \times 22\, mod\, 97 = 3,$$

thus safely obtaining the message $(x = 3)$ sent by the Sender. Note that the next message the Sender sends, the value k will be a different one.

2.3 Hash Functions and Digital Signatures

A ciphered data stream sent through a public channel, may be intercepted and partially or totally changed, or a malicious third party may try to impersonate a Sender. In order to detect any of these cases Digital Signatures (DS) and Hash functions are used.

With Digital Signatures it is possible to authenticate a message, thus guaranteeing that it comes from a specific Sender. The most commonly used algorithms to implement these signatures are the RSA, the ElGamal and the Digital Signature Algorithm (DSA). The signature scheme consist of two components, the message signing ($sig(x)$), performed with an asymmetrical algorithm, and the message validation, accomplished with the use of hash functions.

The asymmetrical algorithms used in the DS schemas are too costly to be directly used in the digital signing of the entire data stream. Another problem would be that each block of the original message would be signed separately; this means that some of the blocks could be removed or replaced without the knowledge of the receiver. The solution lies in the usage of Hash functions, which create a small message that is dependent on the all data stream that has to be signed/verified. The fingerprint (the Digest Message) generated by these algorithms is non-reversible, meaning that the original plain text cannot be retrieved from the message digest. The most commonly used hash functions are the MD5 and SHA. The Whirlpool hashing algorithm is also presented in this thesis, being one of the most promising hash functions for the future. Since the MD5 hash function has been broken, and collision attacks are computationally feasible [29], no work on this algorithm will be presented in this thesis.

When the original fingerprint (DM) of a data stream can be securely procured, for example if stored on a trusted memory, a simplified schema to digitally sign a data stream can be devised to validate its authenticity. Since the original DM of the data stream does not have to be retrieved through an unsecured public channel, asymmetrical algorithms do not have to be used. In this case the Digital Signature of a given data stream can be accomplished simply by the generation of its DM and comparison with the original DM, stored on a trusted location. This is the case in the hardware attestation proposed in Chapter 5. With this simplified approach no asymmetrical based algorithms have to be used, making the validation of a given data stream, a computationally lighter process.

2.3.1 SHA Hash functions

Currently, the most commonly used hash functions are the MD5 and the Secure Hash Algorithm (SHA), with 128-bit to 512-bit output Digest Messages, respectively. While for MD5, collision attacks are computationally feasible on a standard desktop computer, current SHA-1 attacks still require massive computational power [30], (around 2^{69} hash operations), making attacks unfeasible for the time being.

In 1993, the Secure Hash Standard (SHA) was first published by the NIST. In 1995, this algorithm was reviewed [31] in order to eliminate some of the initial weakness. This revised algorithm is usually referred to as SHA-1 (or SHA128). SHA-1 quickly found its way into all major security applications, such as SSH, PGP, and IPSec. In 2001 a new SHA-2 hashing algorithm was proposed. In 2002, the SHA-2 [32] was released as an official standard; it uses larger Digest Messages, making it more resistant to possible attacks and allowing them to be used with larger data streams, up to 2^{128} bits in the case of SHA512. The SHA-2 hashing algorithm is the same for the SHA224, SHA256, SHA384, and SHA512 hashing functions, differing only in the size of the operands, the initialization vectors, and the size of the final digest message.

SHA128 Hash function

The SHA-1 (or SHA128) produces a 160-bit message digest (the output hash value) from the input message. The input data stream is separated into multiple input blocks of 512 bits each. The input block is split into 80x32-bit words denoted as W_t, one 32-bit word for each computational round of the SHA-1 algorithm, as depicted in Figure 2.10. Each round comprises additions and logical operations, such as bitwise logical operations (f_t) and bitwise rotations to the left ($RotL^i$). The calculation of f_t depends on the round t being executed, as well as the value of the constant K_t. The 80 rounds of the SHA-1 are divided into four groups of 20 rounds each. Table 2.5 presents the values of K_t and the logical function applied to each group. The $\wedge$ symbol represents the bitwise AND operation and $\oplus$ represents the bitwise XOR operation. To better illustrate the algorithm, a pseudo code representation is depicted in Figure 2.11. The initial values of the A to E variables in the beginning of each data block calculation correspond to the value of the current 160-bit hash value, DM_0 to DM_4. After the 80 rounds have been computed, the A to E 32-bit values are added to the current Digest Message (or hash value). The Initialization Vector (IV) or the hash value for the first block is a predefined constant value. The

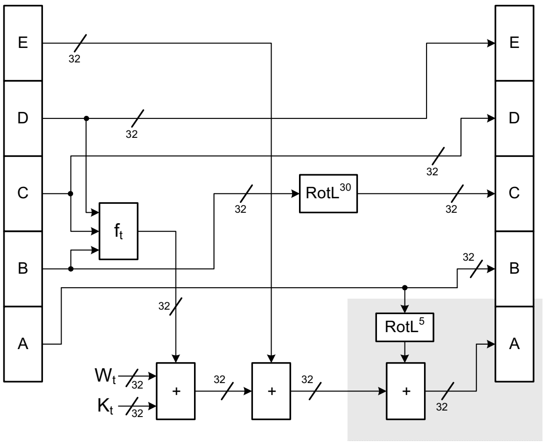


Figure 2.10: SHA-1 Round calculation.

Table 2.5: SHA-1 functions and constants.

Rounds (t)	Function (f_t)	K_t
0 to 19	$(B \wedge C) \oplus (\overline{B} \wedge D)$	0x5A827999
20 to 39	$B \oplus C \oplus D$	0x6ED9EBA1
40 to 59	$(B \wedge C) \oplus (B \wedge D) \oplus (C \wedge D)$	0x8F1BBCDC
60 to 79	$B \oplus C \oplus D$	0xCA62C1D6

output value is the final DM, after all the data blocks have been computed. In some higher level applications such as the keyed-Hash Message Authentication Code (HMAC) [33] or when a message is fragmented, the initial hash value (IV) may differ from the constant specified in [31].

SHA256 Hash function

In the SHA256 hash function a final DM of 256 bits is produced. The input blocks have 512 bits each. Each input block is expanded and fed to the 64 rounds of the SHA256 function in words of 32 bits each (denoted by W_t). In each round of the SHA-2 algorithm the input data is mixed with the current state. Like in the SHA-1, the data scrambling is performed according to the computational structure depicted in Figure 2.12 by additions and logical operations, such as bitwise logical operations and bitwise rotations. The computational structure of each round of this algorithm is depicted in Figure 2.12. Each W_t value is a 32-bit data word and K_t is the 32-bit round dependent

```
for each data_block do
    $W_t = expand$(data_block)
    A = $DM_0$ ; B = $DM_1$ ; C = $DM_2$ ; D = $DM_3$ ; E = $DM_4$
    for t= 0, t≤79, t=t+1 do
        Temp = $RotL^5$(A) + $f_t$(B,D,D) + E + $K_t$ + $W_t$
        E = D
        D = C
        C = $RotL^{30}$(B)
        B = A
        A = Temp
    end for
    $DM_0$ = A + $DM_0$ ; $DM_1$ = B + $DM_1$ ; $DM_2$ = C + $DM_2$
    $DM_3$ = D + $DM_3$ ; $DM_4$ = E + $DM_4$
end for
```

Figure 2.11: Pseudo Code for SHA-1 function.

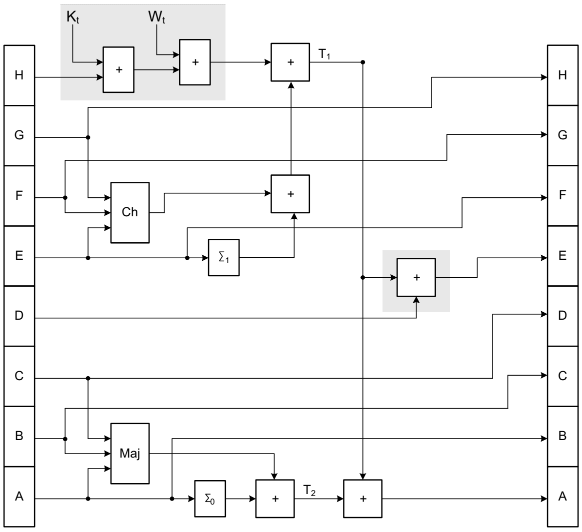


Figure 2.12: SHA-2 round calculation.

constant.

The operations presented in Figure 2.12 for the SHA-2 round are described in Table 2.6. The logical operations Ch, Maj, Σ_i, and σ_i are presented, where $ROTR^n(x)$ represents the right rotation operation by n bits, and $SHR^n(x)$ the right shift operation by n bits.

The 32-bit values of the A to H variables are updated in each round and the

Table 2.6: SHA256 and SHA512 logical operations.

Designation	Function
Maj(x,y,z)	$(x \wedge y) \oplus (x \wedge z) \oplus (y \wedge z)$
Ch(x,y,z)	$(x \wedge y) \oplus (\overline{x} \wedge z)$
$\sum_0^{\{256\}}(x)$	$ROTR^2(x) \oplus ROTR^{13}(x) \oplus ROTR^{22}(x)$
$\sum_1^{\{256\}}(x)$	$ROTR^{14}(x) \oplus ROTR^{18}(x) \oplus ROTR^{41}(x)$
$\sigma_0^{\{256\}}(x)$	$ROTR^7(x) \oplus ROTR^{18}(x) \oplus SHR^3(x)$
$\sigma_1^{\{256\}}(x)$	$ROTR^{17}(x) \oplus ROTR^{19}(x) \oplus SHR^{10}(x)$
$\sum_0^{\{512\}}(x)$	$ROTR^{28}(x) \oplus ROTR^{34}(x) \oplus ROTR^{39}(x)$
$\sum_1^{\{512\}}(x)$	$ROTR^{14}(x) \oplus ROTR^{18}(x) \oplus ROTR^{41}(x)$
$\sigma_0^{\{512\}}(x)$	$ROTR^1(x) \oplus ROTR^8(x) \oplus SHR^7(x)$
$\sigma_1^{\{512\}}(x)$	$ROTR^{19}(x) \oplus ROTR^{61}(x) \oplus SHR^6(x)$

new values are used in the following round. The IV for these variables is given by the 256-bit constant value specified in [31], being set only for the first data block. The consecutive data blocks use the partial DM computed for the previous data block. Each SHA256 data block is processed in 64 rounds, after which the values of the variables A to H are added to the previous DM in order to obtain a new value for the DM. To better illustrate this algorithm a pseudo code representation is depicted in Figure 2.13. The final DM for a given data stream is given by the result obtained from the last data block.

Comparing Figures 2.10 and Figure 2.12, is it noticeable an increase in complexity from SHA-1 to SHA-2.

SHA512 Hash Function:

The SHA512 hash algorithm computation is identical to that of the SHA256 hash function, differing only in the size of the operands, 64 bits instead of the 32 bits of SHA256. The size of the DM and the width of the Σ functions, described in Table 2.6, have 512 bits. Table 2.6 also presents the functions σ used in the data block expansion. The values W_t and K_t are 64 bits wide and each data block is composed of 16x64-bit words, having in total 1024 bits.

```
for each data_block i do
    $W = expand$(data_block)
    A = $DM_0$ ; B = $DM_1$ ; C = $DM_2$ ; D = $DM_3$
    E = $DM_4$ ; F = $DM_5$ ; G = $DM_6$ ; H = $DM_7$
    for t= $0$, t$\leq$ $63$ $\{79\}$, t=t+1 do
        $T_1 = H + \sum_1(E) + Ch(E, F, G) + K_t + W_t$
        $T_2 = \sum_0(A) + Maj(A, B, C)$
        $H = G ; G = F ; F = E ;$
        $E = D + T_1$
        $D = C ; C = B ; B = A$
        $A = T_1 + T_2$
    end for
    $DM_0$ = A + $DM_0$ ; $DM_1$ = B + $DM_1$
    $DM_2$ = C + $DM_2$ ; $DM_3$ = D + $DM_3$
    $DM_4$ = E + $DM_4$ ; $DM_5$ = C + $DM_5$
    $DM_6$ = D + $DM_6$ ; $DM_7$ = E + $DM_7$
end for
```

Figure 2.13: Pseudo Code for SHA-2 algorithm.

Data block expansion for SHA function:

As previously referred, the total amount of data that needs to be fed to the computational rounds of the SHA algorithm is larger than the amount of data of each input data block. Consequently, the input data block has to be expanded.

The SHA-1 algorithm computation steps described in Figure 2.10 are performed 80 times (rounds). Each round uses a 32-bit word obtained from the current input data block. Since, each input data block only has 16x32-bits words (512 bits), the remaining 64x32-bit words are obtained from the data expansion. This expansion is performed by computing (2.25), where $M_t^{(i)}$ denotes the first 16x32-bit words of the i-th data block.

$$W_t = \begin{cases} M_t^{(i)} & 0 \leq \mathrm{t} \leq 15 \\ \\ RotL^1(W_{t-3} \oplus W_{t-8} \oplus W_{t-14} \oplus W_{t-16}) & 16 \leq \mathrm{t} \leq 79 \end{cases} \tag{2.25}$$

For the SHA-2 algorithm, the computation steps shown in Figure 2.12 are performed for 64 rounds (80 rounds for the SHA512). In each round, a 32-bit word (or 64-bit for SHA512) from the current data input block is used. Once more, the input data block only has 16x32-bits words (or 64-bit words for SHA512), resulting in the need to expand the initial data block to obtain the remaining words. This expansion is performed by the computation described in (2.26), where $M_t^{(i)}$ denotes the first 16 words of the i-th data block and the operator $+$ describes the arithmetic addition operation.

$$W_t = \begin{cases} M_t^{(i)} & , 0 \leq \mathrm{t} \leq 15 \\ \sigma_1(W_{t-2}) + W_{t-7} + \sigma_0(W_{t-15}) + W_{t-16} & , 16 \leq \mathrm{t} \leq 63 \ \{\text{or } 79\} \end{cases} \tag{2.26}$$

Message padding:

In order to assure that the input data block is a multiple of 512 bits, as required by the SHA-1 and SHA256 specification, the original message has to be padded. This message padding also comprises the inclusion of the original message size into the padded message, which can be used to validate the original message size. For the SHA512 algorithm the input data block is a multiple of 1024 bits.

The padding procedure for a 512 bit input data block is as follows: for an original message composed of n bits, the bit "1" is appended at the end of the message (the $n+1$ bit), followed by k zero bits, were k is the smallest solution to the equation $n + 1 + k \equiv 448 \ mod \ 512$. In other words the last block is padded with zeros until only 64 bits are left to be filled. These last 64 bits are filled with the binary representation of n, the original message size. This operation is better illustrated in Figure 2.14 for a message with 536 bits (010 0001 1000 in binary representation).

$$\underbrace{10\cdots\cdots01}_{\text{first data block}} \quad \underbrace{\overbrace{10\cdots01}^{\text{24bits}}\,1\,\overbrace{00\cdots00}^{\text{423bits}}\overbrace{0\cdots0\,01000011000}^{\text{64bits}}}_{\text{last data block}}$$

Figure 2.14: Message padding for 512 bit data blocks.

The SHA512 message padding differs in the fact that, each data block has 1024 bits and that the last 128, not 64 bits, are reserved to be filled with the binary value of the original message. This message padding operation can be efficiently implemented in software with a minimal cost.

2.3.2 Whirlpool Hash Function

Whirlpool is a collision-resistant 512-bit hash function operating on messages up to 2^{256} bit long. It is based on a Miyaguchi-Preneel hash compression function and on the Rijndael algorithm [23]. Initially proposed in 2000, it became a part of the ISO/IEC standard and the NESSIE project, after the Whirlpool last revision in 2003.

The core computation of the Whirlpool hash function consists of 3 computation layers and one key addition, repeated for 10 rounds. The 512 bit input value is treated has a 8 by 8 byte matrix. The first computation layer is the non-linear layer (γ), where a non-linear byte substitution is performed for each byte of the 512 bit input. This substitution can be performed by an S-Box (identically to the AES) or in a finer grained computation by 5 4-bit lookup tables and bit wise XOR operation [34] as depicted in Figure 2.15. The Substitution operation is the same for every byte of the 512-bit input. In the second computation layer, the cyclical permutation π, the replaced bytes of layer γ are rotated. The bytes in column j are rotated downwards by j positions. Since these permutations are fixed, they can be hardwired in a hardware implementation. The third computational layer is the diffusion layer θ. In this layer, the linear mapping $\mathcal{M}_{8\times 8}[GF(2^8)] \longrightarrow \mathcal{M}_{8\times 8}[GF(2^8)]$ is performed, where a modular multiplication by a constant polynomial over the Galois Field (2^8) is applied to each row of the input matrix. The calculation of the first output byte (b_{00}) from θ is presented in (2.27), where a_{0i} are the output bytes from layer γ.

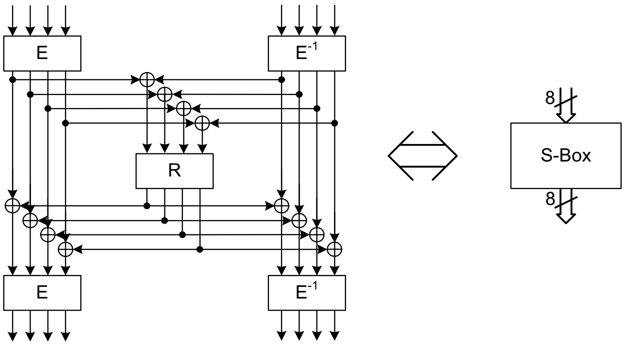


Figure 2.15: Whirlpool S-Box.

$$\begin{aligned} b_{00} = & a_{00} \oplus [a_{01} \otimes 09] \oplus [a_{02} \otimes 02] \oplus [a_{03} \otimes 05] \\ & \oplus [a_{04} \otimes 08] \oplus a_{05} \oplus [a_{06} \otimes 04] \oplus a_{07} \end{aligned} \tag{2.27}$$

Finally the expanded key is added, also in the GF(2^8), corresponding to the XOR bitwise logical operation. These operations are depicted in Figure 2.16.

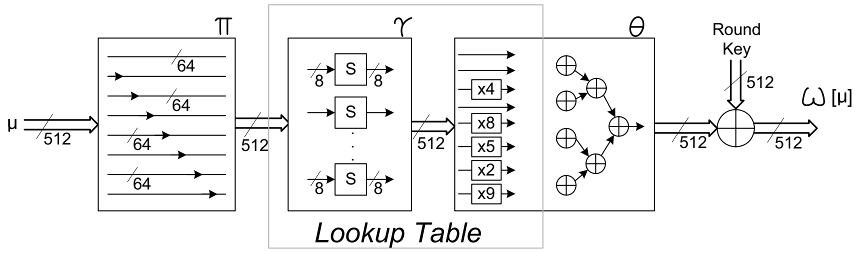


Figure 2.16: $w[\,]$ Whirlpool operations.

The output of one round is used as the input of the next round. After 10 rounds the output value is added, also in the GF(2^8), to the partial digest message, thus generating the new partial digest message H_i, has presented in (2.28); D_i is the i-th 512-bit block of the input message. The final digest message is given by the last partial digest message (H_t) after all input blocks have been processed.

$$\begin{aligned} H_i &= W[D_i \oplus H_{i-1}] \oplus D_i \oplus H_{i-1} \ ; \\ DM &= H_t \ . \end{aligned} \tag{2.28}$$

Key schedule expansion:

As depicted in Figure 2.16, each round of the DM calculation requires a round key. The key schedule expands the initial 512-bit key, also designated by Initialization Vector (IV), into a sequence of round keys K_i^j where j is the round and i corresponds to the data block being computed. This key expansion is performed by applying the Whirlpool core computation to the key, as depicted in (2.29). The input value for each round is the value of the partial digest message H_i calculated in (2.28), except for the first round, where the IV is used. In the Whirlpool standard the IV is a string of 512 bits equal to zero. The Whirlpool key or IV is a known constant value.

$$\begin{aligned} K_0^0 &= IV \ ; & & \\ K_i^0 &= H_{i-1} & & \text{for } i \geq 1 \ ; \\ K_i^j &= W[K_i^{j-1}] & & \text{for } j \geq 1 \ . \end{aligned} \tag{2.29}$$

For the round keys computation, in the key expansion calculation, a predefined constant value C^r is used. Each round has a different constant value [34]. Thus for the computation of each round, two $w[\,]$ operations have to be performed, as depicted in Figure 2.17.

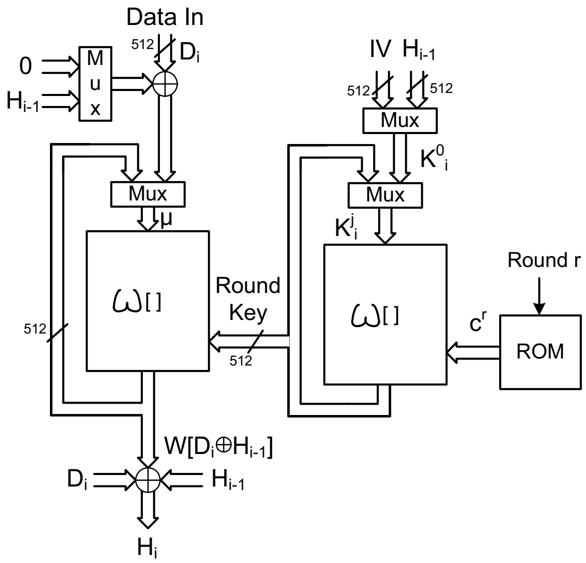


Figure 2.17: Whirlpool hash computation.

Message padding:

To guarantee that an input message, with a maximum length of 2^{256}, is a multiple of 512 bits, the original message is padded. This padding is done by putting a '1' after the last used bit followed by as few '0' as necessary to have the last block with only 256 free bits. These last 256 bits are used to add to the padded message the binary representation of the length of the original message. This is identical to the message padding previously explained for the SHA functions (see page 39). The data blocks are viewed as byte arrays by sequentially grouping the bits in 8-bit chunks.

2.3.3 RSA signature scheme

The RSA signature scheme is implemented with the RSA cipher algorithm. The signature is generated simply by encrypting the DM with the Private Key ($PK = (p, a)$). Since, only the person who is signing the DM knows this key, no one else will be able to generate a DS that when decrypted results in the original DM. The signing algorithm is given by:

$$s = sig_{PK}(x) = x^a \, mod \, n \; , \tag{2.30}$$

where x is the DM of the message being digitally signed.

In order to validate this signature, the known public key ($pK = (p, b)$) is used to decrypt the signature. The signature verification is performed by:

$$ver_{pK} = true \Rightarrow s^b \, mod \, p = (sig_{PK}(x))^b \, mod \, p. \tag{2.31}$$

RSA signature example: Suppose a Sender, with the private key $SPK = (p, a)$ and the public key $SpK = (p, b)$, wishes to send a signature of x to a Receiver with the private key RPK and the public key RpK. Since the signature is being sent through a public channel, it to has to be encrypted (2.32).

$$y = sig_{SPK}\left(e_{SpK}(x)\right) \tag{2.32}$$

At the reception the Receiver decrypts the message with is private key and then verifies the signature with the Sender's public key (2.33)

$$x^{'} = ver_{SPK}\left(d_{SpK}(y)\right) \tag{2.33}$$

The Sender's signature is only validated if $x^{'} = x$.

2.3.4 ElGamal signature scheme

The ElGamal signature scheme uses the same mathematical principals of the ElGamal public-Key cryptosystem. This is a non-deterministic scheme, meaning that several signatures may exist for a given message, thus the verification algorithm has to be able to accept all the possible signatures, since all of them are valid. In the ElGamal signature scheme, the private key ($PK = (p, a)$) and public key ($pK = (p, \alpha, \beta)$) of the ElGamal public-Key cryptosystem can be used. Along with the private and public key, a random number k is also generated, with:

$$0 \leq k < p - 1\ . \tag{2.34}$$

The signing algorithm is given by:

$$sig(x)_{PK} = (\gamma, \delta), \tag{2.35}$$

were

$$\gamma = \alpha^k \, mod \, p \tag{2.36}$$

and

$$\delta = (x - a\gamma)k^{-1} \, mod \, (p - 1). \tag{2.37}$$

In order to validate the signature (2.38) has to verify:

$$ver_{pK} = true \Rightarrow \beta^{\gamma}\gamma^{\delta} \, mod \, p \equiv \alpha^x \, mod \, p\ . \tag{2.38}$$

2.3.5 Digital Signature Algorithm

Digital Signature Algorithm (DSA) [35], specified in the Digital Signature Standard (DSS), is a modification to the ElGamal signature scheme. With the modification on the DSA, it is possible to have a 320-bit signature for a 160-bit message while keeping the same security as the ElGamal signature scheme with 512-bit. The computations in the DSA are still performed using a modulus p with 512 bits. This is only used for digital signatures; it is not used for key exchange.

The private key for this algorithm is $PK = (p, q, \alpha, a)$ and the public key is $pK = (p, q, \alpha, \beta)$. p is a random prime number, with $2^{L-1} < p < 2^L$ for $512 \leq L \leq 1024$ and L is a multiple of 64. q is a random prime divisor of $p - 1$, with $2^{159} < q < 2^{160}$, such that $g = (p-1)/q$ is an integer number. These values can be obtained by choosing a random 160-bit prime number (q) and choosing another random number (g) such that $p = h \times q + 1$ is a prime number with $2^{L-1} < p < 2^L$.

The value α is given by:

$$\alpha = h^g = h^{\frac{p-1}{q}} \tag{2.39}$$

with $1 < h < p - 1$ and $\alpha > 1$. The value of β is given by:

$$\beta = \alpha^a \, mod \, p \tag{2.40}$$

were a is a random value, with $0 < a < q$. In this algorithm the DS of x is given by:

$$sig(x)_{PK} = (\gamma, \delta), \tag{2.41}$$

were

$$\gamma = (\alpha^k \, mod \, p) \, mod \, q \tag{2.42}$$

with k being a random number with $0 < k < q$, and

$$\delta = (x + a\gamma)k^{-1} \, mod \, q. \tag{2.43}$$

In order to validate the signature the following equation has to verified:

$$ver_{pK} = true \Rightarrow (\alpha^{e_1}\beta^{e_2} \, mod \, p) \, mod \, q = \gamma \tag{2.44}$$

were:

$$e_1 = x\delta^{-1} \, mod \, q \tag{2.45}$$

and

$$e_2 = \gamma\delta^{-1} \, mod \, q \tag{2.46}$$

In the Digital Signature Standard, the message to be signed is obtained from the usage of the SHA hash function, used to generate the DM.

2.4 Conclusions

In this chapter, an overview of the algorithms used in current cryptographic systems is presented. A clear computational difference between the three classes of algorithms can be inferred. While the hashing functions deal with relatively simple operations over small operands, the asymmetrical algorithms require mathematically heavy calculations over considerably large operands. These characteristics are reflected in how the algorithms are used. Typically, hash functions are used to reduce the input data stream into a small Digest Message, later encrypted by the asymmetrical algorithm. The symmetrical algorithms are significantly less complex in terms of computation, when compared to the asymmetrical algorithms; therefore, they are the preferred choice when large data streams have to be ciphered.

From the analysis of several encryption algorithms, the following conclusions regarding the choice of the most appropriate algorithms for the Secure Computing Module are devised. To implement the Secure I/O and the Sealed Storage features of Secure Computing, symmetrical algorithms should be used, since large data streams have to be ciphered. For the internal attestation of the execution path, Digital Signatures are needed. However, since the expected data stream (i.e. the binary code of the data path) is known, and given that its DM can be securely stored in the Secure Computing Module ROM, a simplified version of DS can be used. This means that only hash functions are required to validate the authenticity of data streams. Thus, no asymmetrical ciphering is needed to implement the core features of the Secure Computing Module.

The following two chapters present the proposed computational structures for symmetrical encryption algorithms and the hash functions to be used in the Secure Computing Module. Asymmetrical encryption algorithms are not an intrinsic part of the Secure Computing Module hardware structure. Notwithstanding, research work on alternative numbering systems, capable of speeding up the calculation of the asymmetrical algorithms, is presented in appendix A. This appendix presents the work developed in RNS, which is mainly focused on the improvement of the multiplication computation, the most critical operation of asymmetric encryption.

Chapter 3

Proposed Symmetrical Encryption Hardware

Contents

In this chapter, computational structures for the two more frequently used symmetrical algorithms are proposed. The first section proposed a DES computational structure. Even though less frequently used in modern applications, the DES algorithm is still used by many, less recent applications. The existence of a DES computational structure is important due to backward compatibility. The second section, proposes a compact AES structure capable of achieving high throughputs with a small device occupation.

3.1 DES

3.1.1 Proposed DES structure

As presented in the previous chapter, the core computation of DES can be summed up to XOR operations, SBOXs, permutations, and word expansions. Since permutations and expansions can be performed by routing, only the XORs, SBOXs, and some glue logic require computational logic. In order to create a compact DES computational core, a fully folded design has been implemented. In each clock cycle, one round of the DES 16 rounds are computed, thus 16 clock cycles are required to compute a 64-bit data block. The used structure is presented in Figure 3.1. In this folded design, some additional logic is required for multiplexing and for round control.

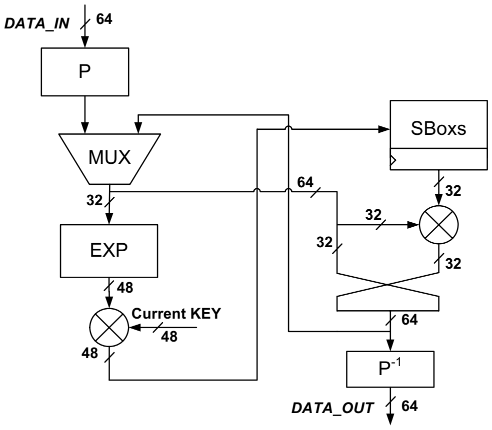


Figure 3.1: DES computational structure.

Given that, this DES core is to be used on an FPGA device, two major computational structures can be chosen for the implementation of the SBOXs. The first and most commonly used is the implementation of the SBOX using the FPGAs Look Up Tables (LUT). In this approach, distributed memory blocks are created for each of the 32 bits of the word resulting from the SBOXs. Since most of the used Xilinx FPGAs have 4 input LUTs, the 6 bit SBOX requires at least 2 LUTs for each output bit. From this, it can be estimated that at least 64 LUTs are required, having a critical path of at least 2 LUTs, as depicted in Figure 3.2.

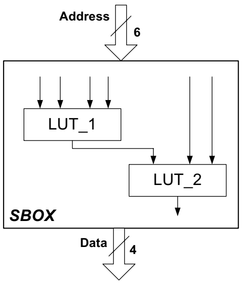


Figure 3.2: LUT based SBOXs.

Taking into account that current FPGAs have embedded memory blocks (BRAMs), an alternative implementation of the SBOXs can be used. These BRAMs can be used as ROM blocks, to implement a full SBOX table. Since these BRAMs have output ports with at least 4 bits, one BRAM can be used to replace at least $2 \times 4 = 8$ LUTs. Moreover, modern FPGAs have embedded dual port BRAMs with more that $(2 \times 2^6 =)$ 128 words, thus, two SBOXs can be computed in each BRAM, as depicted in Figure 3.3. With this, only 4 BRAMs need to be used, instead of at least 64 LUTs. Due to the fact that existing BRAMs have registered output ports the round register must be located at the end of the SBOXs, limiting the options of the designer where to place the round registers.

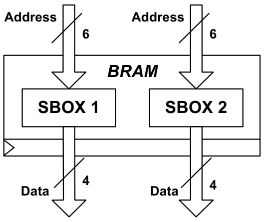


Figure 3.3: BRAM based SBOXs.

The encryption and decryption of data, in the DES algorithm, differs only in the order in which the key expansion is performed. The key expansion consists of fixed permutations and rotate operations. While the permutation operations can be performed by routing, the rotation requires dedicated hardware. The rotation can be of 1 or 2 positions and, depending on the operation (encryption or decryption), to the left or to the right. The implemented structure is depicted in Figure 3.4.

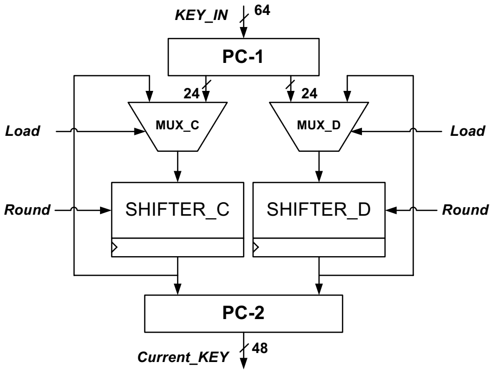


Figure 3.4: DES key expansion.

In order to simplify the computational structure and the key expansion, only the DES algorithm is performed in hardware. To compute the 3DES algorithm, the DES hardware is called 3 times, with the 2 or 3 different keys, thus performing the 3DES calculation.

3.1.2 Performance analysis and DES related work

To evaluate the advantages and disadvantages of using BRAMs on DES computational structures and the polymorphic DES implementation, a Xilinx Virtex II Pro 30 prototyping FPGA device has been used.

In Table 3.1, the two implemented DES computational structures, with and without BRAMs, are compared. In this table, related DES stand-alone art is also presented.

From the implementation results of the proposed DES core with and without BRAMs on the Virtex-2 and Virtex-2 Pro FPGA technologies, it can be concluded that a significant reduction on the required slices (37%), at the expense of 4 BRAMs, can be achieved. However, as a consequence, the critical path

Table 3.1: Stand-alone DES performances

	Wee [36]	Rouv [37]	Our-BRAM	Our-LUT
Device	V2-4	V2-5	V2-5	V2-5
Freq. (MHz)	179	274	152	202
Slices	382	**189**	**175**	**278**
BRAMs	0	0	4	0
Thrput (Mbit/s)	716	974	609	808
Latency	16	18	16	16
TP/S	1.87	5.15	3.48	2.91

	CAST [38]	Our-BRAM	Our-LUT
Device	V2P2-7	V2P30-7	V2P30-7
Freq. (MHz)	261	218	287
Slices	**255**	**175**	**278**
BRAMs	0	4	0
Thrput (Mbit/s)	1044	872	1148
Latency	16	16	16
TP/S	**4.09**	**4.98**	**4.13**

increases about 32%. This delay increase is due to the fact that a BRAM has a critical path equivalent to about 3 LUTs, while the critical path of a LUT implemented SBOX is of 2 LUTs. Nonetheless, an improvement of 20% to the Throughput per Slice (TP/S) efficiency metric can be achieved. In these technologies, and for the BRAM based structures, the slice occupation (2%) is the same as the BRAM usage (2%), thus an adequate utilization of the available resources in the device is achieved. In older technologies, where BRAMs are not so fast, like the Virtex-E, the penalty on the delay is higher. In this case, practically no improvement to the TP/S is achieved (only 2%).

When compared with related art, that uses the unmodified DES algorithm structure, the proposed core has an equivalent TP/S as the commercial core from CAST, when compared with the proposed LUT based DES structure. The TP/S metric improves to 22% when compared with the BRAM based DES structure. When compared with [36] a TP/S metric improvement of 86% and 57% is achieved for the proposed structure with and without BRAMs, respectively.

In [37], the authors propose a modification to the DES computational algorithm, which allows for the efficient use of a pipeline computation, resulting in a very efficient computational structure. This improvement comes at the expense of a higher latency and a potentially lower resistance to side-channel attacks, since the same key is added at two locations, instead of one [39, 40].

This algorithmic alteration also makes the usage of side-channel defenses more difficult [41,42]. Nevertheless, when no side-channel concerns exist, this structure is quite advantageous.

Taking into account that, the computational block used to perform the SBOXs operation is exactly the same in both papers; the same tradeoff between LUTs and BRAMs can still be applied to the design proposed in [37]. As a result, the 64 Slices [37] required for the SBOXs can be replaced by 4 BRAMs, further improving the TP/S efficiency metric, as suggested by the results in Table 3.1.

3.2 AES

In most of the current communication systems, privacy is a key requirement, which is typically achieved by the use of several encryption systems. In 2001, the National Institute of Standards and Technology (NIST) accepted the Rijndael algorithm as the Advanced Encryption Standard (AES) [12, 23]. This new AES has been introduced as the replacement for the old, but still used, Data Encryption Standard (DES) [22]. Even though the AES is one of the most computationally efficient encryption algorithms, it is still very computationally demanding, and not able to achieve the throughput required by some applications when implemented in software. Motivated by the need of higher throughput, several hardware designs of the AES algorithm have been proposed, either for very high throughputs [43–45] or for devices with more limited resource [44, 46, 47], achieving lower throughputs. However, these approaches implement the AES algorithm using fine grained structures, requiring more hardware resources with a more complex structure, that reflects on a lower performance. This paper proposes a coarse grained AES design, employing the FPGA internal memories.

Unlike other designs that only use the FPGAs internal memories to implement the byte substitution operation, this proposal uses these memory blocks to merge the byte substitution and the polynomial multiplication. This memory based structure allows an efficient AES encryption and decryption core implementation. In this thesis, two structures for the AES core are presented: a fully folded one for area constrained implementations, and a fully unfolded structure meeting higher throughput requirements. Both AES cores have low hardware complexity and short critical paths. The proposed design allows a high throughput with low pipeline latency.

3.2.1 Proposed AES structure

In a fine grained implementation of the AES algorithm, both the byte substitution (SBox) and the column multiplication would require specific and distinct hardware structures. According to (2.3), four structures are required per column. Implementing this structure in a LUT based architecture has a significant cost, not only due to the computational units, but also in the resources used in the interconnection. In programmable devices, a significant improvement can be achieved by merging all computations into a single (on-chip) memory block [48]. Such a coarse grained solution is possible due to the fact that the individual resulting coefficients of the polynomial depend only on one of the bytes of the data block (or state matrix). In devices embedding true dual port memory blocks, such as the BRAM in the Xilinx FPGAs, 2 byte substitutions and 2 full multiplications can be mapped into a single memory block, as depicted in Figure 3.5. Each full multiplication multiples one byte ($S_{i,c}$) by a set of constants: $\{1, 1, 2, 3\}$ for the encryption and $\{9, e, b, d\}$ for the decryption.

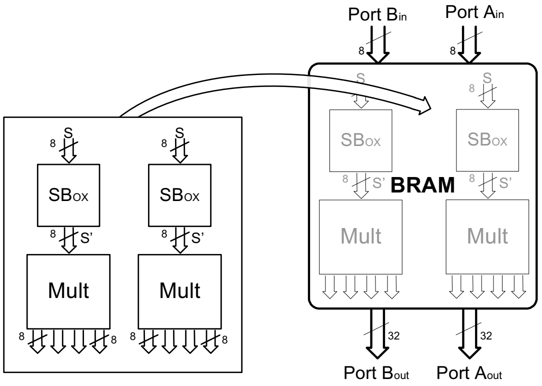


Figure 3.5: Coarse grained column computation using BRAM

Since the byte permutation required in the row shift is a fixed operation, it is performed simply by routing the values to the respective memory block. Only the remaining additions have to be performed in a fine grained organization (e.g. in LUTs). For each byte in a given column, 4 additions have to be performed: 3 to add the polynomial coefficients and 1 for the key addition. It should be noted that, because the byte substitution operation does not depend on the bytes position in the state matrix, the ShiftRow operation can be performed before the SubBytes (see pseudo-code in Figure 2.5, on page 23).

Last round calculation:

The last round of the AES computation has the particularity of not computing the polynomial multiplication, as illustrated in the pseudo-code in Figure 2.5. This can be computed with the memory structure previously presented, which only performs the byte substitution operation. Also, the output value is directly added to the key since, no polynomial addition has to be performed. For reconfigurable area efficiency, the last round is computed using the same memory blocks as the inner rounds. In the encryption, the byte substitution can be obtained directly since this value is equal to the multiplication by 1, given by the memory computation. In the decryption, however, all four results are multiplied by coefficients that are different from 1 (i.e. $\{9, e, b, d\}$). In order to obtain the original value before the multiplication, the logical operation $1 = b \oplus d \oplus e \oplus 9$ can be performed on the four 8-bit outputs of the memory blocks, at the cost of three 8-bit XOR gates, thus obtaining the multiplication by 1.

In order to simplify the selection logic between the encryption and decryption operation, it can be noticed that for the encryption, the XORing of the multiplication coefficients also yields the multiplication by 1, i.e. the logical operation $1 \oplus 1 \oplus 2 \oplus 3 = 1$. Since this operation is performed for both encryption and decryption, no selection logic has to be used, reducing the critical path and the required area resources.

Encryption and decryption AES rounds:

As previously mentioned, the two major differences between the encryption and decryption algorithms are the byte substitution operation and the polynomial multiplication coefficient values. In the proposed memory based implementations, these two differences are located in the memory blocks. Thus, by changing the lookup values in the memories, the computation can change either to encryption or to decryption. For better hardware efficiency, both encryption and decryption can be merged into a single memory block [49]. The encryption/decryption memory block requires 2048 addressable bytes ($2 \times 32 \times 2^8 = 1024$ bytes). The new memory address is given by a byte, of the state matrix, and an additional bit, indicating whether the operation is encryption or decryption, as depicted in Figure 3.6.

The last difference between the encryption and the decryption process, resides in the byte permutation performed by the shift row transformations, as depicted in Figures 3.7. The corresponding byte permutation can be selected by multiplexing the two possible values. After the byte permutation, only half of the new byte positions are different (depicted by the shaded rows in Figure 3.7). Thus, each resulting column calculation only requires the multiplexing of half

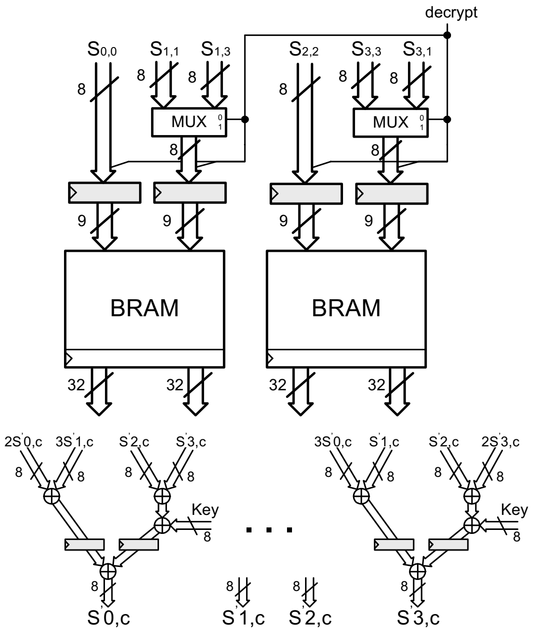
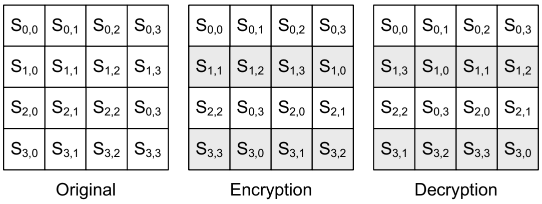


Figure 3.6: AES partial encryption and decryption round

of the byte values to select between encryption and decryption. This is performed by the multiplexer units depicted in Figure 3.6



$S_{0,0}$	$S_{0,1}$	$S_{0,2}$	$S_{0,3}$
$S_{1,0}$	$S_{1,1}$	$S_{1,2}$	$S_{1,3}$
$S_{2,0}$	$S_{2,1}$	$S_{2,2}$	$S_{0,3}$
$S_{3,0}$	$S_{3,1}$	$S_{3,2}$	$S_{3,3}$

Original

$S_{0,0}$	$S_{0,1}$	$S_{0,2}$	$S_{0,3}$
$S_{1,1}$	$S_{1,2}$	$S_{1,3}$	$S_{1,0}$
$S_{2,2}$	$S_{0,3}$	$S_{2,0}$	$S_{2,1}$
$S_{3,3}$	$S_{3,0}$	$S_{3,1}$	$S_{3,2}$

Encryption

$S_{0,0}$	$S_{0,1}$	$S_{0,2}$	$S_{0,3}$
$S_{1,3}$	$S_{1,0}$	$S_{1,1}$	$S_{1,2}$
$S_{2,2}$	$S_{0,3}$	$S_{2,0}$	$S_{2,1}$
$S_{3,1}$	$S_{3,2}$	$S_{3,3}$	$S_{3,0}$

Decryption

Figure 3.7: Byte permutation in the row shifting

The AES core

To compute the AES algorithm, the round calculations have to be performed a predefined number of times depending on the key size. This can be done, either by having one hardware structure per each AES round or by reusing the hardware of the rounds. The first approach is referred to as unfolded loop and the second one, a fully unrolled loop approach by folded loop.

Unfolded loop AES core:

When a high throughput is required, pipelined versions of the fully unrolled approach can be used (referred to as AES unfolded core). In the AES unfolded core, each round is computed in its own dedicated structure. Such a structure can be implemented by connecting the output of one round to the input of the following round, as depicted in Figure 3.8. Since the reconfigurable devices internal memory blocks are synchronous and have registered outputs, the inter-round pipeline consists of the synchronous memory blocks output. Additional pipelining can be introduced, by inter-round registers. These registers can be easily introduced after the memory blocks and anywhere in the additional logic, depicted in Figure 3.6 by the shadowed registers.

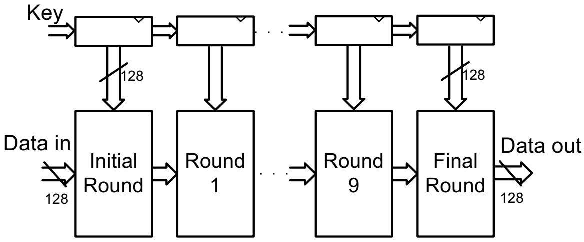


Figure 3.8: AES unfolded core

Folded loop AES core:

When the throughput requirements are not critical or the available hardware is limited, folded versions of the core can be used. In the fully folded design, the throughput is significantly lower, due to the reuse of the hardware structure and due to a longer critical path. The longer critical path is imposed by the additional multiplexer unit required to reinsert the value of the previously calculated round or the new data block, as depicted in Figure 3.9. The primary design trade-off is between performance and hardware resources. The rolled structure can also be advantageous when encryption modes, other than EBC, are used, such as CBC or CFB [50]. In these modes, the next data block can only be encrypted after the previous one has been coded. For these modes, the existence of a pipeline becomes inefficient, since it would be almost always empty. The folded structure, on the other hand, has the round hardware always in use. Figure 3.10 depicts a variation of the folded design to encrypt and decrypt in ECB and CBC modes. Since the difference in the AES algorithm for the 128, 192 and 256 is the amount of computed rounds, the folded AES core

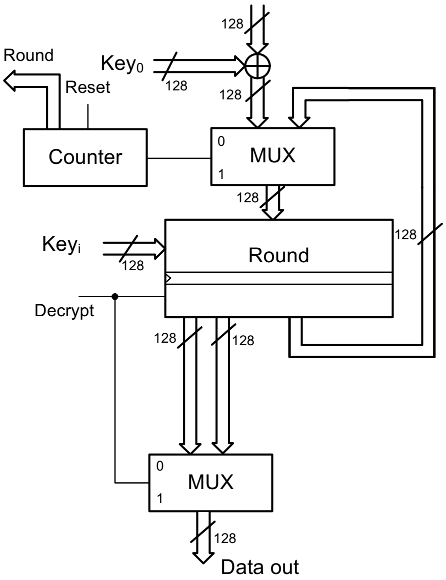


Figure 3.9: AES folded core

is able to cipher in these 3 modes, simply by switching the reset value of the internal counter (to 10, 12, or 14).

Key register:

In the folded design, the expanded key is only accessed in blocks of 128 bits, which is the corresponding size for each round. In the case of the folded design, the key register can be accessed as an addressable register bank, implemented with memory blocks. To better use the dual encryption/decryption capability of the AES, the key register (implemented with memory blocks) can store both the encryption and the decryption keys. Like in the round calculation, the differentiation of the cipher mode is done by adding the encryption/decryption signal to the key register address. In devices with memory blocks having less than 128-bit output ports, the memory based register has to be implemented with several memory blocks. Figure 3.11 depicts such an implementation for a XILINX FPGA using 32-bit output BRAMs. Each 32-bit port comes from a BRAM output port.

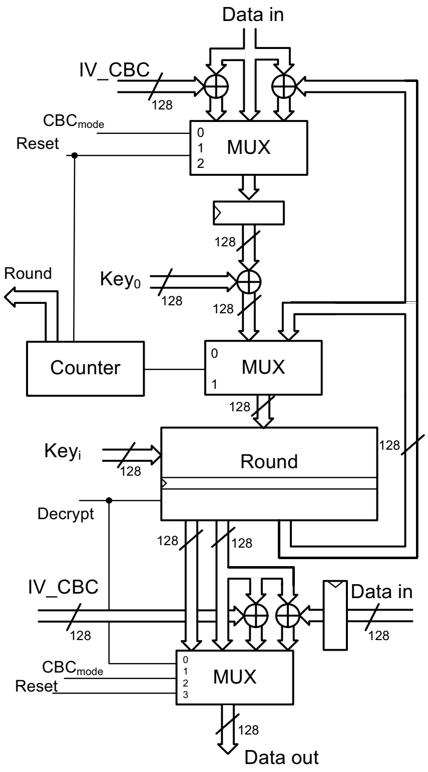


Figure 3.10: AES folded core with ECB and CBC

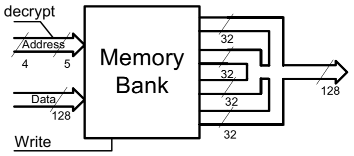


Figure 3.11: AES folded key register

3.2.2 Performance analysis and AES related work

This section evaluates the proposed AES implementations also presenting comparative analysis to related work. The proposed AES structures were implemented on a Xilinx Virtex II Pro (xc2vp20-7) on an Alpha Data: (ADM-XPL) development board using the ISE (6.3) tools from Xilinx.

Table 3.2 presents the implementations results (after Place & Route) of the unfolded AES, with intra-pipelining (intra-pp); and without intra-pipelining (inter-pp), and the folded AES core. Note that all the proposed cores are capable of performing both encryption and decryption.

Table 3.2: AES implementation results

	Slices	BRAM	(MHz) Freq.	(Gbit/s) ThrPut	(cycles) Latency
folded	515 (5%)	12	182	2.3	10
unfolded (inter-pp)	3168 (32%)	80	156	20.0	10
unfolded (intra-pp)	3513 (36%)	80	271	34.7	30

Due to the usage of the internal BRAMs, it is possible to obtain very high throughputs for a relatively low FPGA area utilization, namely a throughput above 34Gbit/s for an 36% and 80 BRAMs occupation of a XC2VP20 FPGA. With the folded design, a throughput of above 2Gbit/s is obtained for an occupation of only 5% and 12 BRAMs, which already includes the additional logic and the 4 BRAMs used to implement the expanded key register. In Figure 3.6, the intra-register pipeline that precedes the BRAM block can be included in the BRAM itself, since these components can be configured to have registered inputs. However, the ISE 6.3, reported worst implementation results for this solution. Since the unfolded AES core is implemented as a sequence of registered rounds, the maximum frequency of the unfolded core without intra-pipelining should be at least as high as the folded AES core. However, even with the additional multiplexors in the folded version the unfolded version is reported to be 16% slower.

In order to analyze the obtained results, the following compares the obtained figures for the fully unfolded design and the fully folded AES structure.

Folded AES core

In Table 3.3, the figures of the proposed folded AES core are compared to the state-of-the-art research and the most recent commercial products. In order to compare the proposed design to the semi unfolded core proposed in [47], ours was unfolded once to obtain an identical throughput. Even though the proposed design requires 2k-Byte RAMs and is focused on a different FPGA technology (a Virtex II Pro), it was implemented on an FPGA with 512-byte internal RAMs. Even in this less favorable case, the proposed AES core outperforms [47] by 560% on the Throughput per Slice metric.

Table 3.3: AES folded core performance comparisons

Architecture	Jhing [47]	Ours	CAST [51]	Helion [44]	Ours
Cipher	Enc./Dec.	Enc./Dec.	Enc./Dec.	Enc.&Dec.	Enc./Dec.
Device	XCV812E	XCV812E	XC2VP20	XC2VP20	XC2VP20
Slices	3046	431	928	1016	515
BRAM	280	32	8	18	12[1]
Freq. (MHz)	61	67	122	198	182
Latency	11	10	11	11	10
Thrput (Gbit/s)	1952	1718	1415	2304	2332
TP/S	**0.6**	**4.0**	**1.5**	**2.3**	**4.6**

When compared to the Helion [44] folded core implemented on a Virtex II Pro, the proposed core outperforms it by 91%. However, the encryption/decryption AES core from Helion, is designed to perform the encryption and decryption calculation, with independent inputs and outputs, which makes it less usable as a single AES core. If these separate inputs/outputs are combined to make a *real* encryption/decryption AES core, the occupation and throughput will most likely be affected, as shown in [45]. It should be noticed that the Helion core has the key expansion in the core itself, while in the proposed approach, the key expansion is performed outside the core.

When compared to one of the leading (real) encryption/decryption AES cores on the market, the CAST [51], implemented on a Virtex II Pro, the proposed AES folded core is over 200% more efficient, regarding the Throughput per Slice metric.

Unfolded AES core

Table 3.4 presents the figures for the fastest high throughput AES cores, including the proposed unfolded AES core. Note that some of the considered cores are only able to perform AES encryption and not AES decryption. In [45], the AES encryption/decryption core is implemented by one encryption unit and another decryption unit. Thus, although not explicitly presented in the paper, the required hardware resources have to be at least twice those of the single encryption core ($2 \times 5408 = 10816$ slices).

The intra-pipelined AES core (Ours-P30), is capable of achieving a 34% faster throughput, than the Giga AES core from Helion [44], which in turn is capable of a maximum throughput of 25Gbit/s. No further comparisons with [44] are be made, since no more figures have been made available by Helion. In implementations, where the output latency is critical, the unfolded AES core without intra-pipelining (Ours-P10) is able to achieve a frequency 24% faster than [45],

Table 3.4: AES unfolded core performance comparisons

	Hodjat [43]	Kotturi [45]	Kotturi [45]	Ours-P10	Ours-P30
Cipher	Encryption	Encryption	Enc./Dec.	Enc./Dec.	Enc./Dec.
Device	XC2VP20	XC2VP70	XC2VP70	XC2VP20	XC2VP20
Slices	5177	5408	10816	3168	3513
BRAM	84	200	400	80	80
Freq. (MHz)	168	233	126	156	272
Latency	41	60	60	10	30
Thrput (Gbit/s)	**21.54**	**29.77**	**16.08**	**19.95**	**34.76**
TP/S	**4.2**	**5.5**	**1.5**	**6.3**	**9.9**

with only 17% of the output latency of [45]. Even when compared to the architectures that only perform encryption, this AES encryption/decryption core outperforms any of the existent cores in speed and in area occupation, resulting in an improvement of the Throughput per Slice metric of almost 100%.

When compared to the existing cores, reported in [45], that are also capable of performing both encryption and decryption, the proposed intra-pipelined core achieves a throughput more than 115% higher, with 68% less reconfigurable area. Thus, achieving a 560% better Throughput per Slice metric, while reducing the output latency by half.

3.3 Conclusions

In this chapter, computational structures for the two most significant symmetrical algorithms, DES and AES, are proposed. The presented encryption hardware has been designed having in mind the Xilinx Virtex II PRO PFGAs; however, the structures proposed are general enough to be used in other reconfigurable technologies as well as in ASIC implementations.

Implementation results, for the proposed DES hardware, suggest that on a Virtex II Pro FPGA, throughputs above 1Gbit/s can be achieved with less than 300 Slices, resulting in a Throughput per Slice metric above 4Mbit/s per slice. It has also been shown that the Throughput per Slice metric can be improved by 20% with the use of BRAMs. The use of the BRAM implies a decrease on the maximum frequency, compensated by a significant reduction on amount of required slices.

The presented hardware AES implementation proposes the improvement of the computational structure by merging the ByteSub operation and the polynomial multiplication operations, and computing them in a single lookup memory bank. With this, significant improvements are achieved. The simplicity of the proposed implementations suggests a high improvement over existing state-of-the-art AES cores, in terms of area and critical path improvement. The later directly reflects to an increase of the AES core throughput. Since the major differences between the encryption and the decryption computations are located in the memory blocks, an AES core employing both encryption and decryption can be implemented at a reduced hardware cost when compared to cores that only implement either encryption or decryption. Another advantage of the proposed design is that, due to the uniformity, simplicity, and regularity of the design, the power consumption is lower and more uniform. The proposed AES core achieves a maximum throughput 34% faster than any known existing AES core, and requires 68% less reconfigurable hardware, with half the pipeline output latency of equivalent cores. These results suggest an improvement on the Throughput per Slice metric of more than 200% for the folded core, when compared to the state-of-the-art research and leading commercial products. The folded AES core is capable of achieving a throughput above 2Gbit/s. For the unfolded structures, the Throughput per Slice metric is improved 560%, achieving a throughput higher than 34Gbit/s.

Chapter 4

Proposed Hash Functions Hardware

Contents

This Chapter is focus on hashing algorithms. These can be extremely useful in data authentication and message integrity checks. Currently, the most commonly used hash functions are the MD5 and the Secure Hash Algorithm (SHA), with 128-bit to 512-bit output digest messages, respectively. Collision attacks have been found for both MD5 and SHA-1 hash functions. While for MD5 it is computationally feasible on a standard desktop computer [29], current SHA-1 attacks still require massive computational power [30], around 2^{69} hash operations, making attacks unfeasible for the time being. For applications that require additional levels of security, the SHA-2 has been introduced. This algorithm outputs a digest message with size from 224 to 512 bits. However the SHA-2 is computational similar to SHA-1 and future attacks can be expected. The mentioned hash functions are also somewhat susceptible to Differential Power Analysis (DPA) attacks [52]. In 2003, the Whirlpool hash function was introduced in the New European Schemes for Signatures, Integrity, and Encryption (NESSIE). This new hash function has a similar security level to SHA-2 (with 512 bits), while at the same time suggesting a better performance [53]. The Whirlpool hash function has also been designed to be more resistant to differential cryptanalysis [34].

Hash functions have the particularity of generating a small fixed length output value, the Digest Message (DM) or hash value, that is highly correlated with the input data, which can be significantly larger (up to 2^{256} bits for Whirlpool hash function). Moreover, the most important characteristics of these functions is the fact that virtually no information about the input data can be obtained from the outputted hash value. The smallest modification in the input data causes a complete modification of the output hash value. Furthermore, the probability of two different input data streams generating the same Digest Message is extremely low.

4.1 SHA

The SHA-1 was approved by the National Institute of Standards and Technology (NIST) in 1995 as an improvement to the SHA-0. SHA-1 quickly found its way into all major security applications, such as SSH, PGP, and IPSec. In 2002, the SHA-2 [32] was released as an official standard. Allowing the compression of inputs up to 2^{128} bits.

As shown in the following section, the SHA computational structures are quite straightforward; unfortunately, the data dependencies of the algorithms do not allow for efficient pipelining. Some work has been done to improve the SHA computational throughput by unrolling the calculation structure, but at the expense of more hardware resources [54, 55].

Efficient hardware implementations of the SHA algorithms are proposed in this section.

Several techniques have been proposed to improve the hardware implementation of the SHA algorithm. The most substantial ones being:

- the usage of parallel counters and balanced Carry Save Adders (CSA), in order to improve the partial additions [56–58];
- unrolling techniques that optimize the data dependency and improve the throughput [54, 57, 59, 60];
- balanced delays and improved addition units, since in this algorithm addition is the most critical operation [56, 59];
- the usage of embedded memories to store the required constant values [61];
- use of pipelining techniques, to achieve higher working frequencies [57, 62].

This work extends the ideas originally proposed by the author in [63, 64], and presents a significant set of experimental results. The major contributions to the improvement of the SHA functions hardware implementation can be summarized as follows:

- Operation rescheduling for a more efficient pipeline usage;
- Hardware reuse in the Digest Message addition;
- A shift based I/O interface;
- Memory based block expansion structures.

The fully implemented architectures proposed in this section, achieve a high throughput for the SHA calculation via operation rescheduling; at the same time, the proposed hardware reuse techniques indicates an area decrease, resulting in a significant increase of the Throughput per Slice efficiency metric.

The following describes the proposed structures for the SHA algorithms.

4.1.1 Proposed SHA-1 Structure

As described in Section 2.3.1, in order to compute the values of one round of the SHA-1 algorithm, the values from the previous round are required. This data dependency imposes sequentiality, preventing parallel computation between rounds. Only parallelism within each round can be efficiently explored. Some approaches [54] attempt to speed-up the processing by unrolling each round computations. However, this approach implies an obvious increase in circuit area. Another approach [55], increases the throughput with the usage of a pipelined structure. Such an approach, however, makes the core inefficient in practical applications, since a data block can only be processed when the previous one has been completed, due to the data dependency of the algorithm.

This thesis proposes a functional rescheduling of the SHA-1 algorithm laid in the work [63], which allows the high throughput of an unrolled structure to be combined with a low hardware complexity, identical to fully folded structures.

Operations rescheduling

From Figures 2.10 and 2.11 in Chapter 2, it can be observed that the bulk of the SHA-1 round computation is oriented towards the A value calculation. The remaining values do not require any computation, aside from the rotation of B. The needed values are provided by the previous round values of the variables A to D.

Given that the value of A depends on its previous value, no parallelism can be directly exploited, as depicted in (4.1).

$$A_{t+1} = RotL^5(A_t) + [f(B_t, C_t, D_t) + E_t + K_t + W_t] \tag{4.1}$$

Nevertheless, since only the operation $RotL^5(A_t)$ depends on the variable A_t, and all remaining terms depend on variables that require no computation and do not depend on the value of A_t, some pre-computation can be performed. In (4.2), the term of (4.1) that does not depend on the value of A, is pre-computed, producing the carry (β_t) and save (S_t) vectors of the partial addition.

$$S_t + \beta_t = f(B_t, C_t, D_t) + E_t + K_t + W_t \tag{4.2}$$

The calculation of A_t, when part of its value is pre-computed during the previous cycle, is described by:

$$\begin{aligned} A_t &= RotL^5(A_{t-1}) + (S_{t-1} + \beta_{t-1}) \\ S_t + \beta_t &= f(B_t, C_t, D_t) + E_t + K_t + W_t. \end{aligned} \tag{4.3}$$

By splitting the computation of the value A and by rescheduling it to a different computational round, the critical path of the SHA-1 algorithm can be significantly reduced. Since the calculation of the function $f(B, C, D)$ and the partial addition are no longer in the critical path, the critical path of the algorithm is reduced to a 3 input full adder and some additional selection logic, as depicted in Figure 4.1.

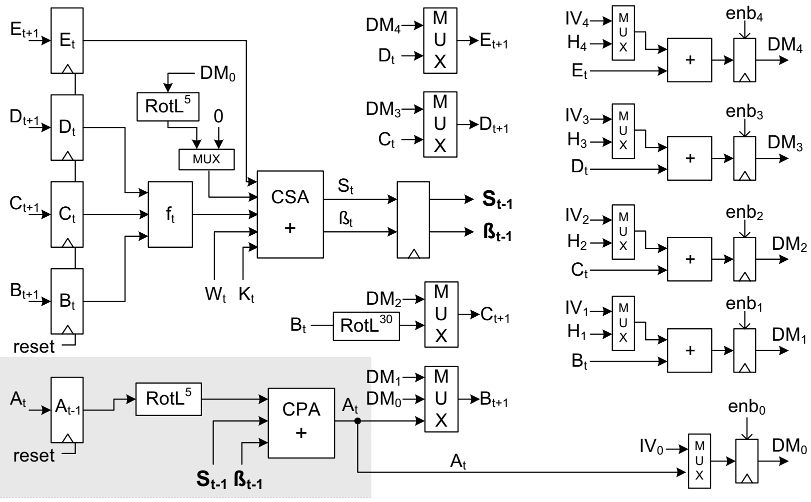


Figure 4.1: SHA-1 rescheduling and internal structure.

With this rescheduling, an additional clock cycle is required, for each data block, since in the first clock cycle the value of A is not calculated (A_{-1} is not used). Note that in the last cycle the values of B_{81}, C_{81}, D_{81}, and E_{81} are not used as well. The additional cycle, however, can be hidden in the calculation of the Digest Message (DM) of each input data block, as explained further on.

After the 80 SHA-1 rounds have been computed, the final values of the internal variables (A to E) are added to the current DM. In turn, the DM remains unchanged until the end of each data block calculation. This final addition is performed by one adder for each 32 bits portion of the 160-bit hash value. The addition of the value DM_0 is directly performed by a CSA adder in the round calculation. With this option, an extra full adder is saved and the DM_0 value calculation, which depends on the value A, is performed in one less clock cycle. Thus, the calculation of all the DM$_j$ is concluded in the same cycle. The additional clock cycle caused by the value A calculation rescheduling becomes hidden.

Hash value initialization

For the first data block, the internal hash value (DM$_0$) has to be initialized. This is accomplished by adding zero to the Initialization Vector (IV). This zero value is generated by resetting the internal registers. This initial value is afterwards loaded to the internal registers (B to E), through a multiplexer. Once again, the initialization of the value A is performed in order to maintain the critical path as small as possible. In this case, the value of DM_0 is not set to the register A. Instead the value of A is set to zero and the value of DM_0 is directly introduced into the value A calculation, as described in (4.4).

$$\begin{aligned} S_0 + \beta_0 &= f(B_{DM_1}, C_{DM_2}, D_{DM_3}) + \\ &\quad + E_{DM_4} + K_0 + W_0 + RotL^5(\mathbf{DM_0}) \\ A_1 &= RotL^5(A_0) + (S_0 + \beta_0) \\ &= RotL^5\,(\,0\,) + (S_0 + \beta_0) \end{aligned} \tag{4.4}$$

The IV can be the constant value defined in [31] or an application dependent value, e.g. from the hashing of fragmented messages. In applications, where the IV is always a constant, the selection between the IV and the current hash value can be removed; a constant value is set into the DM registers. In order to minimize the power consumption, the internal registers are disabled when the core is not being used.

Improved hash value addition

As mentioned earlier, after all the rounds have been computed for a given data block, the internal variables have to be added to the current DM. This addition can be performed with one adder per each 32 bit of the DM, as depicted in the left side of Figure 4.1. In such structure, the addition of the values B through E with the current DM requires 4 additional adders. However, some hardware reuse can be obtained, by analyzing the data dependency and by exploiting the fact that most of the internal variables do not require any computation, since their values are copied directly from the previous values. Taking into account that:

$$E_t = D_{t-1} = C_{t-2} = RotL^{30}(B_{t-3}), \tag{4.5}$$

the computation of the DM from the data block i can be calculated from the internal variable B, as:

$$\begin{aligned} DM4_i &= RotL^{30}(B_{t-3}) + DM4_{i-1}; \\ DM3_i &= RotL^{30}(B_{t-2}) + DM3_{i-1}; \\ DM2_i &= RotL^{30}(B_{t-1}) + DM2_{i-1}; \\ DM1_i &= B_t + DM1_{i-1}. \end{aligned} \tag{4.6}$$

Thus, the calculation can be performed by just a single addition unit and a multiplexer unit, used to select between the value B and its bitwise rotation, $RotL^{30}$. The $rot()$ function in (4.7) represents the optional rotation of the input value.

$$DM[j]_i = rot(B_{t-j+1}) + DM[j]_{i-1} \qquad ;\ 1 \leq \mathrm{j} \leq 4. \tag{4.7}$$

The alternative hardware structure for the addition of the values B to E with the current DM is depicted in Figure 4.2.

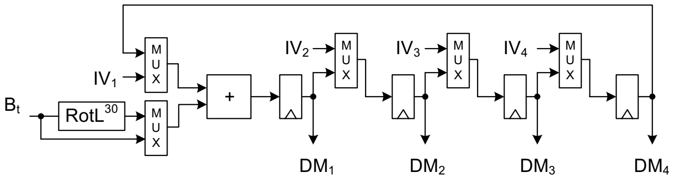


Figure 4.2: Alternative SHA-1 DM addition.

SHA-1 data block expansion

As previously mentioned, the 512 bits of each data block has to be expanded in order to have the required 80 32-bit words (W_t). Since this expansion has to be performed for each data block, see (2.25), it is more efficient to perform this operation in hardware. The input data block expansion described in Section 2.3.1, can be implemented with registers and XOR operations. Finally, the output value W_t is selected between the original data block, for the first 16 rounds, and the computed values, for the remaining rounds. Figure 4.3 depicts the implemented structure. It should be noticed that part of the delay registers have been placed after the calculation, in order to eliminate this computation from the critical path, since the value W_t is connected directly to the SHA-1 core. The one-bit rotate-left operation can be implemented directly in the routing process, not requiring additional hardware.

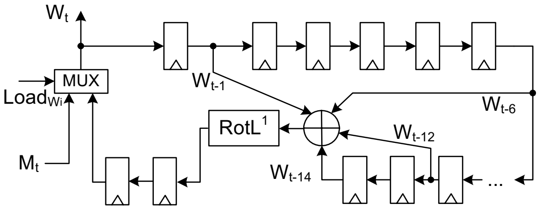


Figure 4.3: Register based SHA-1 block expansion.

4.1.2 Proposed SHA-2 Structure

Like for the SHA-1, the functional rescheduling can also be applied to SHA-2 algorithm. However, as depicted in Section 2.3.1, the SHA-2 computational path is more complex and with an even higher data dependency. In each round of the algorithm, the values A through H have to be calculated, but only the values A and E require computation, since the remaining values are copied directly from the previous round).

Operation rescheduling

In the proposed SHA-2 computational structure [64], it has been identified the part of the computation of a given round t that can be computed ahead in the previous round $t-1$. Only the values that do not depend on the values computed in the previous round can be precomputed. Unlike the rescheduling technique proposed for the SHA-1 algorithm, where the inter round data dependency is low, in the SHA-2 algorithm the data dependency is more complex. While the variables B, C, D, F, G, and H are obtained directly from the values of the round, not requiring any computation, the values of A and E require computation and depend on all the values. In other words, the values A and E for round t cannot be computed until the values for the same variables have been computed in the previous round, as shown in (4.8).

$$\begin{aligned} E_{t+1} =& D_t+\Sigma_1(E_t)+ Ch(E_t,F_t,G_t)+ H_t+ K_t+ W_t \\ A_{t+1} =& \Sigma_0(A_t)+ Maj(B_t,C_t,D_t)+\Sigma_1(E_t) \\ &+ Ch(E_t,F_t,G_t)+ H_t+ K_t+ W_t \end{aligned} \tag{4.8}$$

Taking into account that the value H_{t+1} is given directly by G_t, which in turn is given by F_{t-1}, the precalculation of H can be given by $H_{t+1} = F_{t-1}$. Since the values K_t and W_t can be precalculated and are directly used in each round, (4.8) can be rewritten as:

$$\begin{aligned} \delta_t =& H_t + K_t + W_t = G_{t-1} + K_t + W_t; \\ E_{t+1} =& D_t + \Sigma_1(E_t) + Ch(E_t, F_t, G_t) + \delta_t; \\ A_{t+1} =& \Sigma_0(A_t) + Maj(A_t, B_t, C_t) + \Sigma_1(E_t) \\ &+ Ch(E_t, F_t, G_t) + \delta_t, \end{aligned} \tag{4.9}$$

where the value δ_t is calculated in the previous round. The value δ_{t+1} can be the result of a full addition or the two vectors from a Carry Save Addition. With this computational separation, the calculation of the SHA-2 algorithm can be divided into two parts, allowing the calculation of δ to be rescheduled to the previous clock cycle, as depicted by the grey area in Figure 4.4. Thus, the critical path of the resulting hardware implementation can be reduced. Since the computation is now divided by a pipeline stage, the computation of the SHA-2 will now require an additional clock cycle, to perform all the rounds. In the case of the SHA256, 65 clock cycles are necessary to calculate the 64 rounds.

Hash value addition and initialization

As in the case of SHA-1, the internal variables of SHA-2 also have to be added to the DM. If this addition is implemented in a straightforward manner, 8 adders would be required, one for each internal variable, of 32 or 64 bits for SHA256 or SHA512, respectively. The experimental results for the SHA-1 implementation, presented in Section 4.1.4, suggest the DM addition with the shift to be more area efficient. Thus, only this approach is studied in the proposed SHA-2 structure for the addition of the internal values with the DM value. Since most of the SHA-2 internal values do not require any computation, they can be directly obtained from the previous values of A and E, as depicted in (4.10).

$$\begin{aligned} H_t &= G_{t-1} = F_{t-2} = E_{t-3}; \\ D_t &= C_{t-1} = B_{t-2} = A_{t-3}. \end{aligned} \tag{4.10}$$

The computation of the DM for the data block i can thus be calculated from the internal variables A and E, as:

$$\begin{aligned} DM7_i = E_{t-3} + DM7_{i-1} \ &; \ DM3_i = A_{t-3} + DM3_{i-1}; \\ DM6_i = E_{t-2} + DM6_{i-1} \ &; \ DM2_i = A_{t-2} + DM2_{i-1}; \\ DM5_i = E_{t-1} + DM5_{i-1} \ &; \ DM1_i = A_{t-1} + DM1_{i-1}, \end{aligned} \tag{4.11}$$

with only two full adders, as depicted in (4.12):

$$\begin{aligned} DM[j+4]_i &= E_{t+j} + DM[j+4]_{i-1} \qquad ; 1 \leq \mathrm{j} \leq 3 \\ DM[j]_i &= A_{t+j} + DM[j]_{i-1} \qquad ; 1 \leq \mathrm{j} \leq 3. \end{aligned} \tag{4.12}$$

The selection of the corresponding part of the DM[j] could be performed by a multiplexer. However, taking into account the values of DM[j] are used sequentially, a shifting buffer can be used, as depicted in the right most part of Figure 4.4. Since the values A_t and E_t require computation and the final value is only calculated in the last clock cycle, the calculation of the values $DM0_i$ and $DM4_i$ is performed in a different manner. Instead of using a full adder, after the calculation of the final value of A and E, the DM is added during the calculation of their final values. Since the value of DM$_{i-1}$ is known, the value can be added during the first stage of the pipeline, using a CSA.

After each data block has been computed, the internal values A to H have to be re-initialized with the newly calculated DM; this is performed by a multiplexer that selects either the new value of the variable or the DM, as depicted in the left most side of Figure 4.4. The values A and E are the exception, since the final value computed for the two variables is already the DM.

In the first round, the values of A to H also have to be initialized. All variables, except A and E, are simply loaded with the values in the DM registers, operation depicted in the leftmost part of Figure 4.4. For the A and E variables, the value are fed through the round logic. In this case, all the variables are set to zero (Reset) except the DM_0 and DM_4 inputs. Thus, the resulting value for the registers A and E will be the initialization values of the DM registers.

In the SHA-2 algorithm standard, the initial value of the DM (loaded in A through H) is a constant value, which can be loaded by using set/reset signals

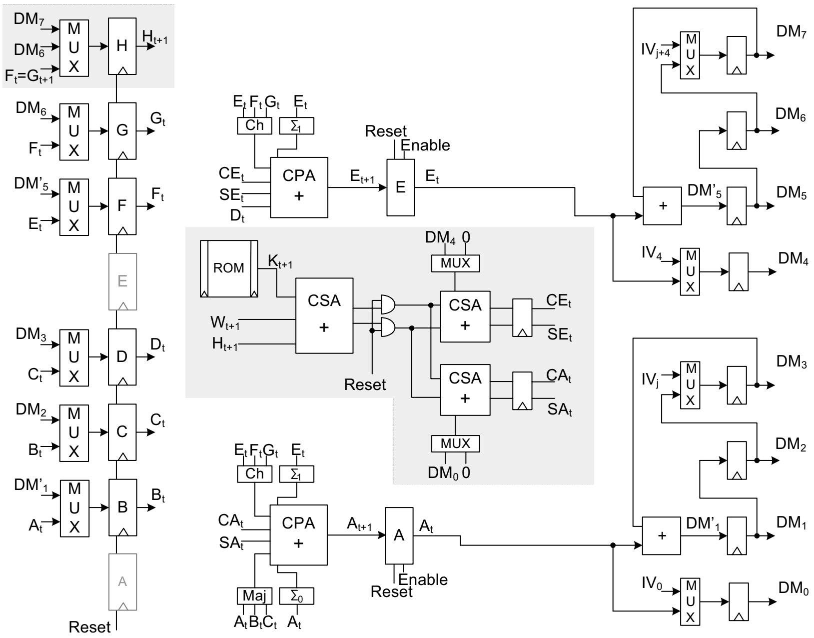


Figure 4.4: SHA-2 round architecture.

in the registers. If the SHA-2 algorithm is to be used in a wider set of applications and in the computation of fragmented messages, the initial Digest Message is no longer a constant value. In these cases, the initial value is given by an Initialization Vector (IV) that has to be loaded. This loading can be performed by multiplexers at the input of the DM registers. In order to optimize the architecture, the calculation structure for the DM can be used to load the IV, not being directly loaded into all the registers. The value of the A and E registers is set to zero during this loading, thus the existing structure acts as a circular buffer, where the value is only loaded into one of the registers, and shifted to the others; this can be seen in the leftmost side of Figure 4.4

This circular buffer can also be used for a more efficient reading of the final DM, providing an interface with smaller output ports. Since less multiplexers have to be used, the values are simply shifted.

SHA-2 data block expansion

As mentioned in Section 2.3.1, the W_t input value has to be expanded according to (2.26). This operation is performed by the data block expansion unit.

The computation structure is similar to the one presented for SHA128. The arithmetic addition represented by the operator $\boxplus$, replaces the bitwise XOR operation ($\oplus$). The proposed structure is depicted in Figure 4.5. The data path is of 32 bits wide for the SHA256 and 64 bits for the SHA512.

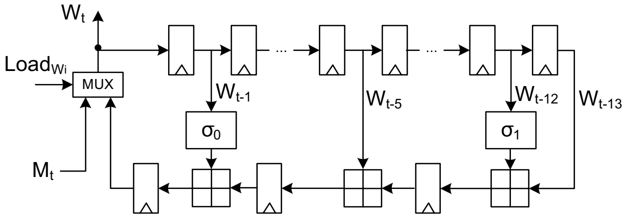


Figure 4.5: SHA-2 data block expansion unit.

4.1.3 SHA implementation

In order to evaluate the proposed designs, the SHA designs have been implemented as processor cores in a Xilinx Virtex II Pro (XC2VP30-7) FPGA using the Xilinx ISE (6.3) and SimplifyPro (8.4) tools. All the values presented in this thesis for the proposed cores were obtained after Place and Route.

In order to fully exploit the capabilities of the reconfigurable device, some design adaptations have been made. The main one lies in the use of fast carry chains for Carry Propagate Adders (CPA) instead of CSA in both pipeline stages. Implementation results showed that a higher performance is achieved using CPA in these FPGA devices. For ASIC implementations, the structures depicted in Figures 4.4 and 4.1 using CSA are more suitable. When implementing the ROM, used to store the K_t values of SHA256 or SHA512, the FPGA embedded RAMs (BRAMs) have been efficiently employed. For the SHA256 structure, a single BRAM can be used, since the 64 32-bits fit in a single 32-bit port embedded memory block. However, for the SHA512 the operands have 64 bits, including the constant K_t. Since the existing BRAMs do not have 64-bit ports, more than one BRAM would be required. However, since BRAMs have dual output ports of 32 bits each, the 80x64-bit constants can be mapped to two 32-bit memory ports; one port addresses the lower 32 bits of the constant and the other, the higher part of the same constant. With this, only one BRAM is used to store the 64-bit K_t constants.

In the FPGA implementation of the data block expansion unit for the SHA-2 algorithm, CPA are also used instead of CSA. The critical path of this unit is of only one CPA, thus smaller than the critical path of the SHA-2 computation core. The use of CSA would require more logic to register both the Carry and Save values. The 4-bit XOR computation in the data block expansion for SHA-1 is also a well suited operation for the 4-bit LUT, present in most Xilinx FPGAs.

Discussion on alternative data block expansion structures

Alternatively to the register based structure presented in Figure 4.3, other structures for the SHA-1 data block expansion can be implemented. One is based on memory blocks addressed in a circular fashion. In the presented implementation, the Virtex II embedded RAMs (BRAMs) are used. The other structure is based on FIFOs.

A 16 word memory is used to store the values with 14 (w_{t-14}) and 16 (w_{t-16}) clock cycles delay. In order to use the dual port BRAMs, the address of the new value has to be the same as the last one, thus the first and the last position of the circular buffer coincide. For this scheme to work properly, the memory must allow for Write after Read (WAR). This however, is only available on the Virtex II FPGA family. In technologies where WAR is not available, the first and last position of this circular memory cannot coincide, thus an additional position in the memory is required as well as an additional port. The W_{t-14} can be addressed by using the $W_{t-16}(= W_t)$ address value subtracted by 2. Identically, the W_{t-3} address can be obtained by subtracting 5 from the W_{t-8} address. The implementation of the 16 bit position circular memory can be done by chaining two 8 positions circular memories, thus requiring less memory for the entire unit, as depicted in Figure 4.6.

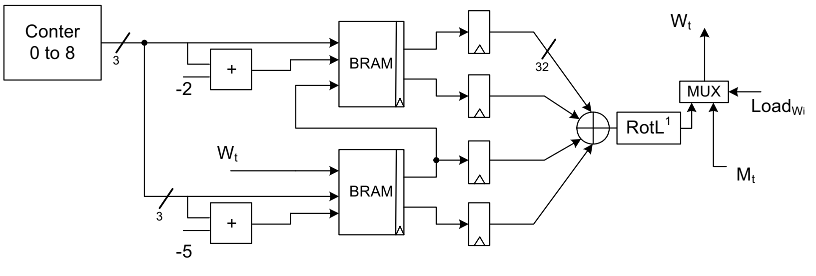


Figure 4.6: BRAM based data block expansion unit.

The data block expansion can also be implemented with FIFOs. In technologies where FIFOs can be efficiently used, the registers used to create the tem-

Table 4.1: SHA-1 data block expansion unit comparison.

Design	Slices	BRAMs
Register based	144	0
Memory based	38	2
FIFO based	100	2
FIFO-MEM based	90	2

poral delay to the value W_t can be replaced by FIFOs. The FIFOs start outputting the values after n clock cycles, where n is the desired delay. The FIFOs have been automatically generated by the *coregen* tool from Xilinx. The resulting computational structure is depicted in Figure 4.7. Circular memories can also be used. For this structure (FIFO-MEM based), counters modulo 5 and modulo 6 have to be used as well as memories that do not require WAR mode.

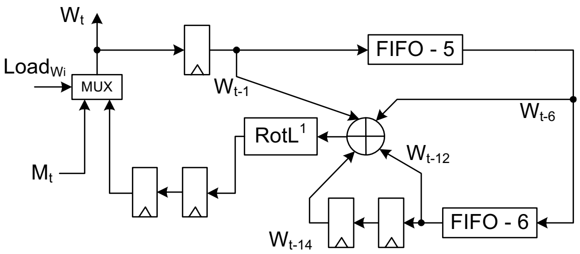


Figure 4.7: FIFO based data block expansion unit.

In order to completely evaluate the proposed structures, the proposed designs have been implemented on a Xilinx Virtex II FPGA. The obtained results are presented in Table 4.1. From this table, it can be concluded that when memories with WAR mode are available, the memory based implementation is more efficient. It requires only 38 Slices and two 32x8 bit memories, resulting in a slice occupation of only 30% of the register based approach, at the expense of 2 BRAMs. However, this is not the case in the majority of available technologies, including most of the existing FPGAs. When memories without WAR mode are available, a reduction of 35% in terms of slice usage can still be achieved, at the expense of 2 embedded RAMs. These data block expansion structures for the SHA-1 algorithm can be directly mapped to the data block expansion of the SHA-2 algorithm.

For the remaining of this thesis, only the register based unit is considered, in order to obtain less technology dependent experimental results.

4.1.4 Performance analysis and SHA related work

In order to compare the architectural gains of the proposed SHA structures with the current related art, the resulting cores have been implemented in several Xilinx devices.

SHA-1 core

In order to compare the efficiency of the DM addition through shift registers proposed in section 4.1.1, the SHA-1 algorithm with variable IV has been implemented with both presented structures. Table 4.2 presents the obtained results for a realization on the Virtex II Pro FPGA. The obtained figures suggest an area reduction of 5% with no degradation on the attainable frequency, resulting in a Throughput per Slice increase from 2.4 to 2.5Mbit/s. In technologies where full addition units are more expensive, like ASICs, even higher improvements are expected.

Table 4.2: SHA-1 DM addition comparison.

Design	traditional addition	shift based addition
Slices	**596**	**565**
Freq. (MHz)	227	227
ThrPut (Mbit/s)	1420	1420
TP/S	**2.4**	**2.5**

The SHA-1 core has also been implemented on a Virtex-E (XCV400e-8) device (Column Our-Exp. in Table 4.3), in order to compare the proposed core with the folded and the unfolded design proposed in [54]. The presented results in Table 4.3 for the Virtex-E device are for the SHA-1 core with a constant initialization vector and without the data block expansion module. When compared with the folded SHA-1 core proposed in [54], a clear advantage can be observed in both area and throughput. Experimentations suggest 20% less reconfigurable hardware and 27% higher throughput, resulting in a 57% improvement on the Throughput per Slice (TP/S) metric. When compared with the unfolded architecture, the proposed core has a 28% lower throughput; however, the unrolled core proposed in [54] requires 280% more hardware, resulting in a Throughput per Slice, 2.75 times smaller than the core proposed in this paper.

Table 4.3 also presents the SHA-1 core characteristics for the Virtex II Pro FPGA implementation. Both the core with a constant initialization vector

Table 4.3: SHA-1 core performance comparisons.

	Lien [54]	Lien [54]	Our–Exp.
	Virtex-E	Virtex-E	Virtex-E
Expansion	no	no	no
IV	cst.	cst.	cst.
Slices	484	1484	388
Freq. (MHz)	103	73	135
ThrPut (Mbit/s)	659	1160	840
TP/S	**1.4**	**0.8**	**2.2**

	CAST [65]	Helion [66]	Our–Cst.	Our+IV
	XCV2P2	XCV2P	XCV2P30	XCV2P30
Expansion	yes	yes	yes	yes
IV	cst.	cst.	cst.	yes
Slices	568	564	533	565
Freq. (MHz)	127	194	230	227
ThrPut (Mbit/s)	802	1211	1435	1420
TP/S	**1.4**	**2.1**	**2.7**	**2.5**

(Our–Cst.) and the one with a variable IV initialization (Our+IV) are presented. These results also include the data block expansion block. The results are compared in Table 4.3 with the related art, including the most recent and efficient commercial SHA-1 cores known by the author.

When compared with the leading commercial SHA-1 core from Helion [66], the proposed structure requires 6% less slices while achieving a throughput 18% higher. These two figures result in a gain on the Throughput per Slice metric of about 29%.

For the SHA-1 core capable of receiving an IV other than the constant specified in [31], a slight increase in the required hardware occurs. This is due to the fact that the IV can no longer be set by the set/reset signals of the registers. This however, has a minimal effect in the cores performance, since this loading mechanism is not located in the critical path. The decrease of the Throughput per Slice metric, from 2.7 to 2.5, caused by the additional hardware for the IV loading is counterbalanced by the capability of this SHA-1 core (Our+IV) to be used in the processing of fragmented messages.

SHA 256 core

The proposed SHA256 hash function structure has been also compared with the most recent and most efficient related art, for cores proposed both in the

Table 4.4: SHA256 core performance comparison.

	Sklav [67]	Our	McEv. [59]	Our	Helion [68]	Our
Device	XCV	XCV	XC2V	XC2V	XC2PV	XC2PV
IV	cst	yes	cst	yes	cst	yes
Slices	1060	764	1373	797	815	755
BRAMS	$\geq$1	1	$\geq$1	1	1	1
Freq.(MHz)	83	82	133	150	126	174
Cycles	n.a.	65	68	65	n.a.	65
ThrPut (Mbit/s)	326	646	1009	1184	977	1370
TP/S	**0.31**	**0.84**	**0.74**	**1.49**	**1.2**	**1.83**

previously published research and the best currently available commercial core, known by the author. The obtained comparison figures are presented in Table 4.4. When compared with the most recent academic work [59, 67] the results show higher throughputs, from 17% up to 98%, while achieving a reduction in area above 25% up to 42%. These figures suggest a significant improvement to the Throughput per Slice metric in the range of 100% to 170%. When compared with the commercial SHA256 core from Helion [68], the proposed core suggests an identical area value (less 7%) while achieving a 40% gain to the throughput, resulting in an improvement of 53% to the Throughput per Slice metric. Note that from the analyzed cores, the proposed core is the only one capable of loading the IV.

In the proposed FPGA implementation the logic required for the IV loading is located between registers as depicted in Figure 4.4. If the IV loading mechanism was not present, the reconfigurable logic located in the CLB of the final register would be unused. Thus, one can state that the IV loading mechanism is implemented with approximately no area cost. Since this loading is performed with only an additional multiplexer, located between registers, it does not influence the critical path of the circuit, as confirmed by the implementation results. The structure proposed by McEvoy [59] also has message padding hardware; however, no figures are given for the individual cost of this extra hardware. This message padding is performed once, at the end of the data stream, and has no significant cost when implemented in software. Thus, the majority of the proposed cores and commercial cores do not include the hardware for this operation.

Table 4.5: SHA512 core performance comparison.

	Sklav [67]	Lien [54]	Lien [54]	Our
	XCV	XCV	XCV	XCV
Expansion	yes	no	no	yes
IV	cst	cst	cst	yes
Slices	2237	2384[1]	3521[1]	1680
BRAMS	n.a.	n.a.	n.a.	2
Freq.(MHz)	75	56[1]	67[1]	70
Cycles	n.a.	n.a.	n.a.	81
ThrPut (Mbit/s)	480	717	929	889
TP/S	**0.21**	**0.3**[1]	**0.26**[1]	**0.53**

	McEv. [59]	Our	Our
	XC2V	XC2V	XC2VP
Expansion	yes	yes	yes
IV	cst	yes	yes
Slices	2726	1666	1667
BRAMS	≥ 1	1	1
Freq.(MHz)	109	121	141
Cycles	84	81	81
ThrPut (Mbit/s)	1329	1534	1780
TP/S	**0.49**	**0.92**	**1.01**

SHA 512 core

Table 4.5 presents the implementation results for our proposed SHA512 core and the most significant related art. The figures suggest a significant reduction to the required reconfigurable area, from 25% up to 60%, while achieving a speed-up of the hashing calculation. When compared with [67], the proposed core requires 25% less reconfigurable logic while a throughput increase of 85% is achieved, resulting in a Throughput per Slice metric improvement of 165%. From all known SHA512 cores, the unrolled core proposed by Lien in [54] is the only one capable of achieving a higher throughput. However, this throughput is only slightly higher (4%), requiring twice as much area as proposed structure. It should also be noticed that the results presented by Lien in [54], marked with ([1]) in Table 4.5, do not include the data expansion module, which would increase the required area. Even in this case the proposed core indicates a 77% higher Throughput per Slice metric. Table 4.5 also suggests a substantial advantage in the proposed core when implemented on the Virtex II Pro (XC2VP), the technology for which the core was originally developed.

4.2 Whirlpool

In this section, a new structure for the Whirlpool hash function implementations is presented. By exploring some of the computational and mathematical characteristics of these algorithms, a faster and more efficient hardware implementation is achieved. A lookup table based Whirlpool implementation is described as well as the computational merging of both the key computation and the data compression.

In order to validate and test the structure, the proposed core [69] has been implemented in a Xilinx Virtex II Pro FPGA. Implementation results suggest a throughput of 5.47Gbit/s with a corresponding Throughput per Slice metric of 2.59Mbit/s per Slice.

4.2.1 Proposed Whirlpool Structure

The following describes the merging on the S-BOXes with the polynomial multiplication into embedded memories and the merging of the round key computation with the data hashing; proposed to improve the Whirlpool computation structure.

Lookup table implementation of Whirlpool: Identically to the AES algorithm, where a more efficient hardware can be achieved by using T-Boxes [70], in the Whirlpool function the non-linear layer γ and the diffusion layer θ can also be merged. This is possible since, the operations are performed at the byte level and, the cyclical permutation layer π can be performed before the non-linear layer γ. With this modification, the main computational of the Whirlpool algorithm, which resides in the substitution operation of layer γ and part of the polynomial multiplication of layer θ, can be computed by lookup tables, as depicted in Figure 4.8. For the 512-bit message, 64 lookup tables of 8-bit

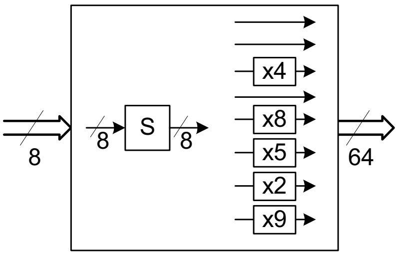


Figure 4.8: Whirlpool Lookup table.

input and 64-bit output are required.

The output of the lookup table are the individual input bytes multiplied by the required coefficients. Thus, to conclude the computation of layer θ, the corresponding bytes of the polynomial multiplication have to be added. This operation is implemented by a tree of XOR gates, described in Figure 4.9 by the θ' box. To complete the Whirlpool core operation, only the round key addition has to be performed.

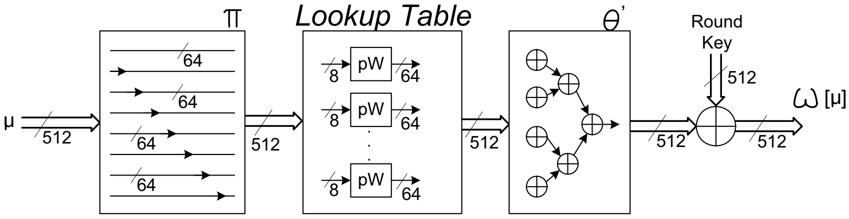


Figure 4.9: W[] operations with Lookup table.

Taking into account that, each of the 64 bytes of the 512 input of a Whirlpool round ($w[\ ]$) require 1 lookup table, a less granular structure can be used, by analyzing and reducing the polynomial expression outputted by the lookup table. The 8 coefficients of the multiplied polynomial are:

$$(01, 01, 04, 01, 08, 05, 02, 09). \tag{4.13}$$

The first and trivial reduction, lays in the removal of the computational redundancy of the three multiplications by 01. The second reduction can be obtained by computing some of the coefficients outside the lookup table. This external computation reduces the size of the lookup table, but has to be properly chosen in order not to significantly affect the critical path of the overall computation. Analyzing the coefficients, it becomes clear that, the most efficient option is the computation of multiplication with the coefficient 02 from the computed multiplication by 01. Identically the computation of multiplication with the coefficient 08 from the computed multiplication by 04. Given that, this multiplication is being performed in $GF(2^8)$, this calculation cannot be performed just by left shifting the two values. Since the $GF(2^8)$ operations are limited to 8 bits, whenever the result goes beyond 8 bits, the remainder polynomial has to be calculated to limit the final result to 8 bits. This polynomial reduction, in the case of the constant multiplication by 2, can be performed by subtracting the irreducible polynomial that defined the field for the Whirlpool, namely the

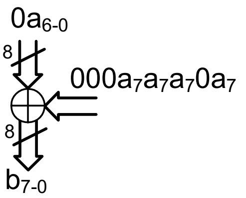


Figure 4.10: $2\times$ Galois Field (2^8) multiplication.

polynomial:

$$p(x) = x^8 + x^4 + x^3 + x^2 + 1. \tag{4.14}$$

Taking into account that, in $GF(2^8)$, this subtraction is performed by the bitwise XOR operation, the multiplication by 2 can by defined as the less significant 8 bits of original value a(x) shifted left one bit and subtracted by the binary value 100011101, when the 8th bit of a(x) is equal to 1, as described in (4.15).

$$\begin{aligned} b(x) = 2 \times a(x) = &(a_{6-0} << 1) && \text{when } a_7 = 0; \\ &(a_{6-0} << 1) \oplus 00011101 && \text{when } a_7 = 1. \end{aligned} \tag{4.15}$$

As depicted in Figure 4.10, (4.15) can be implemented with a single XOR gate in the critical path and a total of 4 XOR gates, per multiplication. This results in a very compact hardware structure, having a negligible impact in the Whirlpool critical path. With the lookup table computing only part of constant multiplications and the presented hardware structure calculating the remaining values, a good compromise can be achieved between a coarse grained and a fine grained implementation of the algorithm. With this, only 32-bit lookup tables are required.

Merging computational blocks: As described in Section2.3.2 (page 39), the core computation of the Whirlpool algorithm is used in the data hashing, but also to compute the round key. This additional computation is advantageous for the safety of the algorithm, also making it more resistant to DPA attacks [34]; consequently, it has additional computational costs that, in a hardware implementation, represent more silicon area. This however, can be minimized, since the computation performed to obtain the round key is exactly the same as the computation for the data compression. By merging this computation into a single hardware structure, a significant gain in terms of area can be

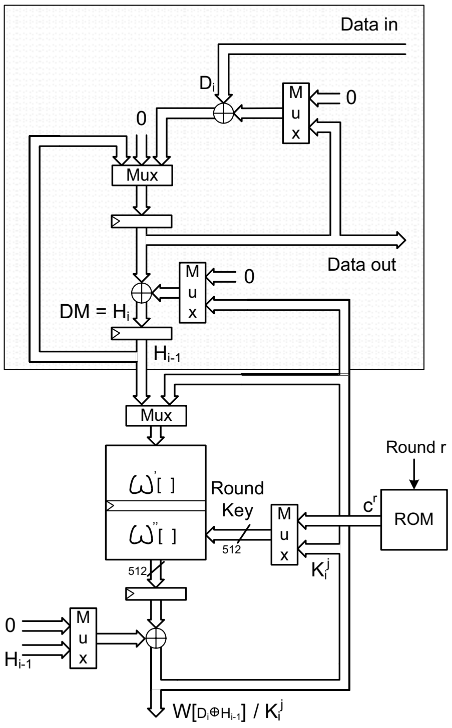


Figure 4.11: Whirlpool proposed core.

achieved. In order to allow both computations to coexist and maintain approximately the same performance, a pipeline structure can be used. By merging the computation, some additional multiplexing logic is required, as depicted in Figure 4.11. With this one level pipeline structure, the 10 computational rounds of the Whirlpool algorithm require 20 clocks cycles. Given that the round key computation uses the partial digest message (H_{i-1}) of the previous round, one additional clock cycle is required. This is caused by the fact that, the next data round key computation can only start when the computation of the previous partial digest message (H_{i-1}) is concluded, the computation of each data block of 512 bits requires in total 21 clock cycles. Note that a pipelining structure would not efficient if the computational merging were not explored, due to the high data dependency of the Whirlpool algorithm, since one round can only be started once the data from the previous round is fully computed.

In order to simplify the data input/output and reduce the amount of required hardware, the input registers described in Figure 4.11 within the grey area, also

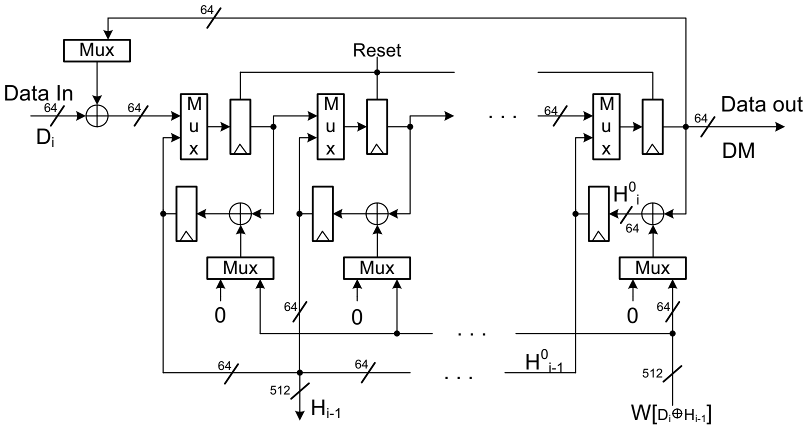


Figure 4.12: I/O registers in the proposed Whirlpool core.

work like shift registers. This is depicted with more detail in Figure 4.12 for a 64-bit data input/output. The $D_i \oplus H_{i-1}$ addition is performed as data are loaded, thus only one XOR gate, with the size of the input bus, is required.

4.2.2 Performance analysis and Whirlpool related work

In order to evaluate and compared with the current Whirlpool state of the art the proposed structure has been realized on a Xilinx Virtex II Pro (XC2VP30-7) FPGA using the Xilinx ISE (6.3) tool. Additional realizations for a Virtex-E and a Virtex-4 have also been obtained in order to properly compare with the existing state of the art cores.

By using the embedded block memories of the Virtex FPGAs (BRAMs), a compact implementation can be achieved. These memories can be directly used to implement the lookup tables described in section 4.2.1. For the Virtex II Pro and Virtex-4 FPGAs, BRAMs can be used with outputs up to 32 bits per BRAM. For the Virtex-E family, which only has BRAMs with outputs up to 16 bits, two block memories have to be used for each lookup table. Taking into account that these BRAMs already have registered outputs, the pipeline depth has to be determined by the registers at the end of the loop computations. Experimental results indicate that the addition of H_{i-1} is better performed before the last register, despite of what is suggested by Figure 4.11. The position of these additions can, and should, be adjusted according to the particular technology used. Given that, the values C^r mapped in the ROM (Figure 4.11) are only 10, and with very few bit lines with logical value '1', it is more efficient to implement this in pure combinational logic and not in BRAM. The limited amount of logic before each register and the structure of the CLBs in the used FPGAs, allow for a compact implementation of the multiplexing structure depicted in Figure 4.12. Also, the use of shift registers for the data input/output, facilitates the routing within the FPGA device. The results for the FPGA implementations are depicted in Table 4.6. This table also presents data for the current state of the art.

Table 4.6: Whirlpool performance comparison

	Kits [1]	Our	Prams [71]	Our	McLo [53]	Our
Device	XCV1000	XCV1000	XC2VP40	XC2VP30	XC4V	XC4V
Slices	5585*	2138	1456	2110	4956	2118
BRAMS	0	64	0	32	68	32
Freq.(MHz)	75*	67	131	224	94	220
ThrPut (Gbit/s)	4.48*	2.38	0.38	5.47	4.79	5.38
TP/S	**0.8**	**1.11**	**0.26**	**2.59**	**0.97**	**2.54**

When compared with the structure proposed in [1], implemented on a Virtex-E FPGA, an improvement of merely 39% is achieved; however, the values presented in the referenced paper are from synthesis only, while ours are after Place&Route. Furthermore, the values presented with a star (*) in Table 4.6 are incoherent when compared with other papers [53, 71] and the presented results. For a very fine grained structure, the number of slices is too small. In addition, the maximum frequency reported in this paper [1] is unrealistically high (75 MHz). For example, the Whirlpool core proposed in this paper has an apparently shorter critical path for the same FPGA technology and employs pipeline techniques, but is only able to achieve 67MHz. These unrealistic results might be due to the inaccuracy of the synthesis estimator. A more detail analysis of the figures presented in [1] is made in Appendix B.

In [71], a compact implementation is presented for a Virtex II Pro FPGA. The proposed Whirlpool structure requires 45% more slices and 32 BRAM, not used in [71]; however, it is capable of achieving a throughput more than 13 times higher. In terms of Throughput per Slice, the proposed Whirlpool structure is 10 times more efficient than the one proposed by Prams [71]. The Throughput per Slice metric does not take into account the BRAM usage, since these are available resources, independently if they are used or not. As BRAMs are hardwire cores in the considered FPGA technology, the silicon area they occupy on the chip is much smaller than the area of the slices. Therefore, introducing the BRAM area into a different throughput per silicon area metric is not expected to introduce significant changes in the figures reported in the last row of Table 4.6.

Compared with the most recent state of the art [53] implemented on a Virtex-4, the comparative results show that the proposed core is capable of a throughput increase of 12%, while reducing by 57% the required slices and 50% the required BRAMs. In terms of global efficiency estimated by the Throughput per Slice metric, the proposed core is 162% more efficient.

4.3 Conclusions

In this chapter the three hashing algorithms most likely to be used in the future were presented, namely the SHA-1, SHA-2, and Whirlpool algorithms. The presented figures, show that adequate throughputs can be achieved with a reasonable reconfigurable area occupation.

Regarding the SHA algorithms, the proposed hardware rescheduling and reutilization techniques improve the hardware realizations, both in performance and in area resources. With the operation rescheduling, the critical path can be reduced in a similar manner as in structures with loop unrolling, but without duplicating the required hardware, leading also to the usage of a well balanced pipeline structure. An efficient technique for the addition of the Digest Message is also presented. This technique allows for a substantial reduction on the required reconfigurable resources and conceals the extra clock cycle delay introduced by the pipeline. The required reconfigurable resources are also significantly reduced due to the way the Digest Message is added to the intermediate values, requiring less multiplexors and adders. By adding and loading the variable A (and E for the SHA-2) through the round hardware, area can also be saved and one less computational cycle is required to add the Digest Message, thus concealing the extra cycle created by the reschedule, which directly reflects on the average throughput of the cores. Two SHA-1 cores have been developed, one that uses a constant IV and a second one that allows different initialization vectors, in order to be used with fragmented messages and facilitate the calculation in HMAC protocols. The core with the IV loading requires some additional hardware for the registers initializations; this however, does not influence the throughput since it is not located in the critical path. The SHA-2 cores have only been developed with variable IV loading.

On a Virtex II Pro FPGA, the proposed SHA-1 and SHA256 cores are capable of a throughput of 1.4Gbit/s, with only 533 and 755 Slices respectively. For the SHA512 core, a throughput of 1.8Gbit/s was obtained using 1667 Slices.

Implementation results clearly indicate significant performance and hardware gains for the proposed cores when compared to the existing commercial cores and related academia art. For the SHA1 implementation, results suggest an improvement of the Throughput per Slice metric of 29% when compared with commercial products and 59% to the current academia art. For the SHA256 the Throughput per Slice metric has been improved by 53% for the considered commercial core and more than 100% when compared with the related academic art.

In the proposed Whirlpool implementation the S-Boxes and part of the GF(2^8) multiplications are merged into a single operation, performed by a lookup table. Since the most complex part of the GF(2^8) multiplication is computed by a lookup table, a faster and more compact structure is achieved. Additionally, the merging of the round key computation and the data compression hardware is also proposed. Implementation results suggest an overall improvement of the Throughput per Slice to related art of about 160%; having a 12% faster throughput with only 43% of the reconfigurable area occupied. On a Virtex II Pro FPGA, a throughput of 5.47Gbit/s is achieved, with 2110 Slices and 32 BRAMs, resulting in a Throughput per Slice of 2.59.

Chapter 5

Attestation of Reconfigurable Hardware

Contents

This chapter is dedicated to dynamic partial reconfiguration of FPGAs, and in particular to the attestation of hardware structures loaded during this dynamic reconfiguration. The attestation module herein proposed is used not only to validate the correctness of the configuration data, but also to protect the remaining computational structures already loaded into the device.

In the Secure Computing context of this thesis, the configuration data cannot be considered trustworthy, since it might be stored on an unsecured location. With this fact comes the need to assure that whatever is being loaded into the reconfigurable device is in fact being loaded to the intended location and that once loaded it will behave as expected.

Taking into account that a hardware structure for a given reconfigurable device can be described by data in binary format, the verification of the hardware structure can be directly performed over the configuration data. This approach allows to perform in an identical manner the hardware and software attestation or in any other digital data stream. In this case, the proof that the configuration bitstream has not been tampered with, and that it is in fact the desired bitstream (i.e. the intended hardware structure), is achieved by computing and comparing the hash value of the data stream. As described before, the Digest Message (DM) of a message, or in this case the configuration bitstream, produces an identifying footprint of the processed data. In these algorithms the probability of two different input data streams generating the same hash value is very low, even if this is done intentionally in order to forge the signature of a given data stream.

This chapter starts by describing the dynamic reconfiguration capability of the FPGAs, and in particular the Virtex II Pro Xilinx FPGA and the main characteristics of the configuration bitstream. Section 5.2 describes the proposed attestation module, detailing the two major operations performed by this module, namely the region delimitation hardware and the hash generation of the bitstream. This chapter is concluded with an overview of the characteristics of the resulting attestation module.

5.1 Dynamic FPGA Reconfiguration

The current generation of reconfigurable devices has the capability to reconfigure part of its available resources while, at the same time, the remaining resources of the device continue active and performing computation. This operation is called dynamic partial reconfigurability. This type of reconfigurability allows for runtime reconfiguration, adaptive hardware algorithms, reduced power consumption, and a more efficient usage of the available resources. For current FPGA devices, data is loaded on a column basis, with the smallest load unit being a frame, which varies in size depending on the targeted device [72].

The two main methods for partial reconfiguration are Difference-based and Module-based. In Difference-based partial reconfiguration, small changes to a design are supported by generating a bitstream based on only the differences in the two designs. Since in the proposed Secure Computing Module (SCM) complete functional units have to be reconfigured, this reconfiguration method is not applicable.

The Module-Based Partial Reconfiguration is the method used in the proposed SCM. In this reconfiguration mode, a fraction of the FPGA is completely reconfigured. For designs where the modules are completely independent (e.g. no common I/O signals except clock signals), bus macros are not needed. However, for modules that communicate with each other, a special bus macro allows signals to cross over a partial reconfiguration boundary. Without this especial feature, inter-module communication would not be feasible, as it is impossible to guarantee routing between modules. The bus macro provides a fixed “bus” of inter-design communication. Each time partial reconfiguration is performed, the bus macro is used to establish unchanging routing channels between modules, guaranteeing correct connections.

The reconfiguration of the Xilinx Virtex II Pro FPGA is performed by the Internal Configuration Access Port (ICAP), whose interface is depicted in Figure 5.1. The FPGA is reconfigured by sending the configuration bitstream through the 8 bit InData port. The configuration interface operates at 50MHz, and for a Virtex II PRO 30 a full configuration bitstream has approximately 1.7 Mbyte; the reconfiguration of the whole device requires approximately 34ms.

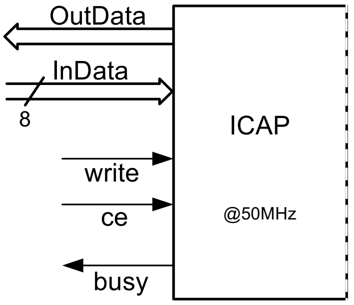


Figure 5.1: Xilinx ICAP interface.

All logic resources encompassed in the reconfigured frames can be changed, including the slices, buffers, block RAMs, multipliers, and I/O ports.

The Virtex Pro devices have an on-chip DES (or Triple-DES) decryption core, used merely to decrypt the incoming bitstreams. The designer can encrypt the bitstream in software, and the FPGA then performs the reverse operation, decrypting the incoming bitstream, and internally recreating the intended configuration. This method provides a very high degree of design security. Without knowledge of the encryption/decryption key or keys, potential attackers cannot use the externally intercepted bitstream to analyze or to clone the design [15].

System manufacturers can be sure that their Virtex-II Pro implemented designs cannot be copied and reverse engineered. However, the main goal of Secure Computing is to protect the user, not the Intellectual Property of the designers. This mechanism will not protect an attack were the bitstream of an encryption unit is replaced by other bitstreams from a different computational units; for example, an older equivalent encryption core with a known backdoor or side-channel weakness. This type of encryption is also not useful for public distribution of computation cores.

The Virtex II Pro devices store the internal decryption keys in a few hundred bits of dedicated RAM, backed up by a small externally connected battery. These bits can only be written, no hardware connection allows them to be read, except by the FPGA's embedded DES core itself. In the new Virtex 5 FPGAs the AES algorithm with a 256-bit key is now used to decrypt the bitstream [73].

The attestation module is used to enforce reliability in the reconfigurable SCM. This attestation procedure is mainly composed by two mechanisms, the Region Delimitation and the Hardware Attestation.

The bitstream has to be interpreted in order to implement the Region Delimitation feature in the proposed attestation module. The internal configuration space of the FPGA is partitioned into segments called "Frames." The portions of the bitstream that actually get written to the configuration memory are "Data Frames." To perform this region delimitation, the bitstream has to be read and interpreted in order to identify which frame of the device is being updated. The following gives an overview on how the device is configured and on the structure of the configuration bitstream.

Virtex II Pro Configuration Registers

The Virtex configuration logic was designed so that an external source may have complete control over the configuration functions by accessing and loading addressed internal configuration registers. In Table 5.1, the most significant internal configuration registers, used in the device configuration, are presented.

Table 5.1: Virtex II Pro internal configuration registers.

Register Name	Description
CMD	Command Register
FLR	Frame Length Register
FAR	Frame Address Register
FDRI	Frame Data Input Register
CRC	Cyclic Redundancy Check
MFWR	Multiple frame Write Register
KEY	Initial Key Address Register
CBC	Cipher Block Chaining Register

These registers provide the following information to reconfigure the FPGA.

Frame Address Register (FAR): The FAR has the frame address for the next configuration data input write cycle. This Register indicates which frame, and where in the frame, the 32-bit data configuration data is to be written. The register is automatically incremented after every written word. The size of a frame is specified in the Frame Length Register.

Frame Length Register (FLR): The FLR is used to indicate the frame size to the internal configuration logic. The value loaded into this register is the number of actual configuration bits that get loaded into the configuration memory frames.

Frame Data Register Input (FDRI): The FDRI forms a pipeline input stage to store the configuration Data Frames in the configuration memory. Starting with the frame address specified in the FAR, the FDRI writes its contents to the configuration memory frames. The FDRI automatically increments the frame address.

Multiple Frame Write Register (MFWR): The Multiple Frame Write Register is used when the bitstream compression option is enabled. When more than one frame has identical data, the Data Frame can be loaded once into the Multiple Frame Write Register then copied into multiple memory address locations.

Initial Key Address Register (KEY): The Initial Key Address Register indicates the number of DES keys that are used for bitstream decryption, in case the bitstream data is encrypted. This is a write-only register; there is no way to read this register through any configuration interface or user logic resources.

Cipher Block Chaining Register (CBC): The Cipher Block Chaining Register stores the DES encryption keys. This is a write-only register; there is no way to read the keys out of the device through any configuration interface or user logic resources.

CRC Register: The CRC is loaded with the Cyclic Redundancy Check (CRC) value that is embedded in the bitstream and compared against an internally calculated CRC value. This value is used to check for errors in the transmission of the bitstream data, and can be easily tampered with. The reset of the CRC register, and associated logic, is controlled by the CMD register.

Command Register (CMD): The CMD is used to execute commands on the device. Table 5.2 presents the most relevant commands available in the FPGA configuration. These commands are executed by loading the CMD register with the respective binary code.

Table 5.2: Virtex II Pro CMD Register commands.

Command	Binary code	Action
RCRC	0111	Reset CRC Register
SWITCH	1001	Change clock frequency
WCFG	0001	Write Configuration Data
RCFG	0100	Read Configuration data
LFRM	0011	Last Frame Write
SHUTDOWN	1011	Begins Shutdown sequence
START	0101	Activates the reconfigurable hardware
MFWR	0010	Activate Multiple Frame Write mode

Bitstream Packets

The configuration bitstream is stored in a file, the bitstream file. Since this file may include a header with information not relevant for the configuration itself (e.g. the file creation date), a synchronization tag is inserted before the actual configuration bitstream. The synchronization tag is also used to synchronize the ICAP with the beginning of the 32-bit packet, since the data is sent to the ICAP interface in blocks of 8 bits, as depicted in Figure 5.1. Consequently, no actual processing takes place until the synchronization tag is detected by the ICAP.

After synchronization, all data (commands, configuration data, etc.) is encapsulated in packets. There are two kinds of configuration packets, Type 1 and Type 2. Type 1 packets are used for register writes. A combination of Type 1 and Type 2 packets is used for Data Frame writes. A packet contains two different sections: Header and Data. A Type 1 Packet Header, shown in Figure 5.2, is always a single 32-bit word that describes the packet type, whether it is a read/write operation, which register is being targeted, and how many 32-bit words are in the following Packet Data portion; a Type 1 Packet Data portion may contain anywhere from 0 to 2047 data words of 32 bits.

Packet Header	**Type**	**Operation (Write/Read)**	**Register Address**	**Byte Address**	**Word Count (32-bit data words)**
Bits	**[31:29]**	**[28:27]**	**[26:13]**	**[12:11]**	**[10:0]**

Figure 5.2: Bitstream Type 1 header.

Type 2 packets must always be preceded by a Type 1 packet that contains no packet data. A Type 2 packet also contains both a header and a data portion, but the Type 2 packet data can be up to 1,048,576 data words in size [15]. The Type 2 packet header, shown in Figure 5.3, differs slightly from a Type 1 packet header, namely because there is no Register Address or Byte Address fields. To write a set of Data Frames to the configuration memory, after the starting frame address has been loaded into the FAR, a Type 1 packet header issues a write command to the FDRI, followed by a Type 2 packet header specifying the number of data words to be loaded, followed by the actual Data Frame packets. Writing Data Frames may require a Type 1/Type 2 packet combination, or just a Type 1 packet, depending on the amount of data to be written.

Packet Header	**Type**	**Operation (Write/Read)**	**Word Count (32-bit data words)**
Bits	**[31:29]**	**[28:27]**	**[26:0]**

Figure 5.3: Bitstream Type 2 header.

Due to the way data is processed by the ICAP, a 32-bit data block is only interpreted after the subsequent 32-bit data block is received. This implies a configuration delay of at least 4 clock cycles and the need to conclude the bitstream with dummy words. This gives time for the attestation module to suspend the reconfiguration process before the activation of the invalid hardware structure by the ICAP.

5.2 Hardware Attestation Module

In order to assure to the user that the cryptographic units in use are the correct ones, an attestation method has to be applied when these units are dynamically allocated into the reconfigurable device. As explained in the next sections, the hardware structure of the unit to be loaded into the device is described by a set of packets. Thus, the validation of this set of packets/instructions (the bitstream) allows the authentication of a given hardware structure, typically by the calculation of the Digest Message of this data stream. This data bitstream can be validated through software or specialized hardware. The existing mechanisms for software attestation could be used [16, 17], however the size of the bitstream can be significantly large, e.g. 1.4Mbit for 10% of the Virtex II Pro 30 FPGA. Therefore, the computational time of this software attestation would also be significant. In software, the bitstream would first need to be loaded into the internal secure memory, validated by a hashing algorithm, and sent to the configuration interface, via the internal memory. Assuming an internal secure memory large enough to store the desired bitstream and that the ICAP was directly connected to the internal memory, a significant computational time would still be required.

In this thesis, the use of hardware structures to perform the attestation is proposed. In this approach, the generation of the Digest Message is performed as the hardware is reconfigured in the device. Given that the hash computation can be performed at a faster rate than device re-configuration (see the implementation results in Chapter 4.1), and that the data can be directly sent to the ICAP, no additional time is required for the attestable reconfiguration, regarding the standard dynamic reconfiguration of the device. However, since the Digital signature of the unit can only be obtained after the complete Digest Message is computed, some region delimitation has to be enforced, to guarantee the non tampering of the remaining area of the circuit. An attacker might create a bitstream that targets the bus responsible for the transmission of the generated DM, putting whatever value he might desire, including that of the correct unit. This would render the attestation module inutile.

To protect the SCM against this type of attacks, the area of the device allowed to be reconfigured is limited to the region reserved for the reconfigurable Cryptographic Computational Units (CrCU). If an attempt is made to modify (write) any of the reconfigurable hardware outside the delimited area, an abort signal is generated and the reconfiguration process stopped. The following describes the hardware structures realized to implement the desired functionalities of the attestation module, namely: *i)* region delimitation; *ii* Hardware validation; *iii)* and a simple interface to use the attestation module.

5.2.1 Region Delimitation

To detect a write violation outside the delimited area, two things must be known: which frame is being written and when is the frame being written. Since the ICAP does not directly provide this information, the bitstream must be interpreted. In order to assure that all the bitstream is analyzed, certain commands cannot be allowed to be processed by the ICAP, such as the *Shutdown* or *the Multiple Frame Write* mode.

To simplify the interpretation of the bitstream, the attestation is organized into several units. First the 8 bit input stream is filtered, in order to eliminate all data before the synchronization sequence and to align the 8-bit input of the ICAP. The beginning of each packet is identified and in order to form the 32-bit words that constitute one packet, a 32-bit register is used. After receiving 4 new 8-bit inputs and the register loaded with a new packet, the output of the register is made available and the $NewPacket$ signal generated, indicating that a new packet is available.

The 32-bit packets, of commands or data, are then sent to the bitstream parser that interprets the bitstream data. This parser mostly implements a state machine that identifies which kind of packet is received and their function. Note that the flow of data into the parser is not constant; a delay of at least 3 clock cycles between each packet exists, since 32-bit packets are processed while 8-bit words are received by the ICAP. This means that the new state of the parser machine is only evaluated when a new packet is received, and the outputs are only updated after each state update.

Given that this unit has to identify where the Data Frame is being written to, writings to the Frame Length Register (FLR), Frame Address Register (FAR), and Frame Data Register Input (FDRI) have to be detected. The parser has its own internal frame register, which is updated every time the FAR packet is sent to the ICAP. The value written to the FDRI is retrieved and used to calculate which frames will be written by the arriving data packets. To assure that the a frame is not improperly written due to a wrong FLR value, when the frame size is sent to the device, via a FLR writing, the value is compared with the correct value, which is known for the device in use. The internal frame register of the attestation is initialized to zero, identically to the FAR register in the FPGA configuration logic. Only when Data frames are sent to the ICAP is the current frame address tested, otherwise during other configuration instructions, an abort signal could be generated. For example, a non initialized FAR register (pointing to zero) would generate an out of bound error for the initial configurations instructions on a valid bitstream sequence.

As mentioned above, it is not sufficient to test this type of instructions; command instructions also have to be interpreted, in order to assure a testable configuration procedure. The following points out which commands are not allowed and why they cannot be executed.

SWITCH : By changing the clock frequency, an operating frequency could be set in which the SCM could behave erroneously.

SHUTDOWN : While in shutdown mode, the configured logic in the device is rendered inoperable; consequently, during this period the attestation Module would by rendered inoperable.

MFWR : The implemented attestation module is only able to cope with a frame write at a time. This means that only one of the frames' writes would be within the Region Delimitation hardware testability. The other frames being written could be located anywhere within the device.

RCFG : By reading the internal configuration of the device, an attacker would have access to critical data, for example to the encryption keys.

The not acceptance of the Read Configuration data command is not compulsory since no hardware structure exists to write the data of the frames back into memory; however, it is recommended as a safety precaution.

Whenever the internal frame address is outside the defined set of allowed frames, or when an illegal command is detected, the $Abort$ signal is generated and the device reconfiguration suspended. In the attestation module output, 4 bits are used to identify what kind of violation generated the $Abort$ signal. Whenever a new packet is made available, an internal counter is incremented, indicating the number of packets received for a given bitstream. The 28-bit value of this counter is used in the validation test.

5.2.2 Hardware Validation

The hardware validation is performed by generating the Digest Message of the data being used to configure the device and comparing it with the expected value. In the proposed attestation module, the DM is computed only with the bitstream used by the ICAP. The data preceding the synchronization pattern is disregarded.

The SHA256 hash function is used to generate the DM in this version of the Secure Computing Module. This algorithm (see Chapter 2.3.1) computes the hashing of a data stream in blocks of 512 bits and requires 65 clock cycles to compute each data block. The SHA256 core at 50MHz would only be able to have a data compression rate of 393Mbit/s, which is lower than the maximum input rate of the ICAP (of 400Mbit/s). If the SHA-1 algorithm were to be used this difference would be even bigger. Rather than having the ICAP wait for the bitstream to be compressed, the hash core is operated at a higher frequency, for example at the frequency of the CrCUs. A FIFO is used to interconnect the bitstream filter, running at the ICAP frequency, with the validation hardware. This FIFO, capable of receiving and sending data asynchronously, acts as a buffer between the two computation units. If the hashing core is not able to process the data at an adequate rate, the FIFO $full$ signal is used to halt the ICAP and the reconfiguration process, while the hashing core processes the data.

Instead of having the parser unit detect the end command in the bitstream, the hash unit is always computing any incoming data. This is used as an additional security measure, since a new bitstream sequence can be added to the end of the configuration file sent to the ICAP. When the reconfiguration unit stops sending data to the ICAP, the GPP of the SCM resumes the computation. Only at this time can one be assured that no more data is sent to the ICAP. At this moment, software code has to "read" the attestation module in order to retrieve the DM and compare it with the expected value in order to determine the authenticity of the loaded hardware.

When the hardware validation unit receives this $read$ signal, two situations can occur: *i)* the configuration bitstream is a multiple of 512 bits and the current DM is the final DM value; *ii)* the configuration bitstream is not a multiple of 512 bits. In this later case, the last partial input of the hashing core is concatenated with zeros, to form a full 512-bit block, and processed to generate the final DM. This concludes the computation of the attestation module.

Once the final DM is generated, the validation data is written into an exchange register (XREG) that connects this module with the GPP. This register and its usage are described in more detail in Chapter 6.1. The computational flow of the attestation module is depicted in Figure 5.4.

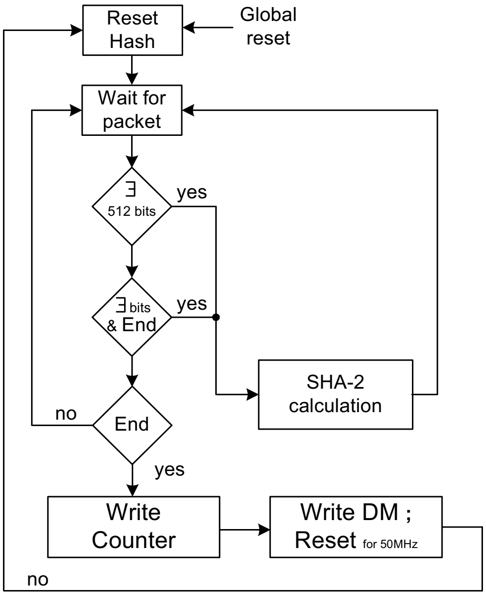


Figure 5.4: Attestation module flow diagram.

Note that, for efficiency reasons, the DM generated with the SHA-2 algorithm does not conform with the standard [32]. The final DM is calculated without the concatenation of the length of the hashed message. The length of the bitstream is considered separately to the hash value. This increases the security level of the validation data, that is composed by the DM, the *Abort value* (if an $Abort$ occurred), and the length of the loaded configuration bitstream.

5.2.3 Attestation Module Interface

In order to facilitate the utilization of this attestation module, an interface identical to the CrCUs is used. In terms of the interface with the GPP, 3 major ports are used: *i)* the $readHash$ signal, that signals the module that there is no more data to be hashed, and to conclude the computation; *ii)* the data output bus, a 32 bit bus connecting the attestation module with the GPP exchange register bank; and the output address port, indicating which of the 32bit parcels of the validation data is being written into the exchange register.

As depicted in Figure 5.4, when the $readHash$ signal is received, any remaining packets are added to the DM. After the final DM value is computed, the validation value is written into the exchange register bank. First the counter value and the $Abort$ signals are written, followed by the 8 32-bit words of the generated DM value.

The counter value is composed of 28 bits, allowing for bitstreams with 256M words; only the words after the synchronization tag are counted. This value is concatenated with the 4 bits of the *Abort value* that indicate if an abort signal was generated and for which reason.

While the DM value is being sent to the GPP, a reset signal is sent to the region delimitation unit and packet counter. This reset initializes the unit for the next bitstream. Once the DM value is loaded into the exchange register, the hashing core is reset and the attestation module goes into the initial state, where it waits for a new bitstream. This means that the attestation module is reset every time the validation data is read.

In order to assure that the part of the attestation module running at 50MHz is reset, the reset signal for these units is set active during the 8 cycles needed to transfer the DM value.

The structure of the attestation module is depicted in Figure 5.5

5.2.4 Implementation Results

The proposed attestation module was implemented on a Virtex II Pro 30 Xilinx FPGA, in order to analyze the amount of resources required to implement this module. The results were obtained after place and route, considering the attestation module as an isolated component.

Implementation results suggest an occupation of approximately 8% of the available slices in the FPGA and 2 BRAMs, for the complete attestation module supported by a SHA256 hashing core. Note that the SHA256 core, by itself, requires 1 BRAM and approximately 900 Slices, i.e. about 7% of the available resources in the FPGA. If the SHA-1 algorithm were to be used to produce the DM, a smaller attestation module would be achieved, since this core only requires 6% of this FPGA device. An attestation module using a SHA-1 core would require approximately 7% of the available slices and 1 BRAM.

The FIFO in the attestation unit was realized using the Xilinx ISE IPCore generator. It requires one embedded BRAM and is capable of storing 128 words of 32 bits. This FIFO also outputs additional status information, e.g.

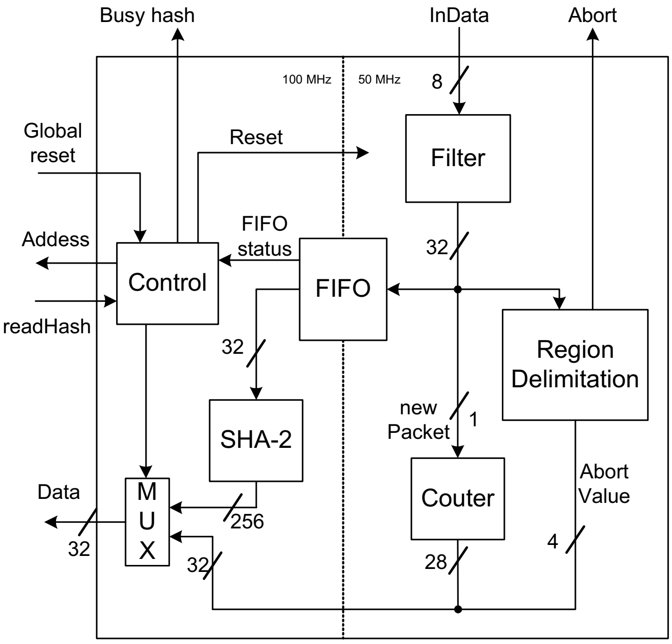


Figure 5.5: Attestation module structure.

full, 512 bits available, and empty FIFO.

5.3 Conclusions

In this chapter, a novel method is proposed to validate the hardware structures loaded into reconfigurable device at run time, via dynamic reconfiguration. This method is based on the analysis of the bitstream used to configure the FPGAs. The attestation of the computational structure being loaded is performed by computing in hardware the Digest Message of the bitstream and comparing it against the expected value. However, this operation is not sufficient to assure the correct loading of the desired structure; the validation of the loaded structure can only be performed after the bitstream has been completely uploaded into the FPGA. An additional mechanism has to be used in order to assure that an adulterated bitstream is not able to damage or modify the Secure Computing Module itself. This is achieved by restricting the area where the reconfiguration can occur. An attempt to modify any resources outside the delimited area, leads to the halting of the reconfiguration. Implementation results suggest that the proposed attestation module, using the SHA-2 algorithm to produce the DM, can be realized using less than 10% of the resources available in the used device. If the SHA-1 algorithm is used, less than 5% of the available resources are needed. This algorithm, however, is not as secure as the SHA-2. With this attestation module, assurance in the dynamic reconfiguration of hardware structures can be accomplished with practically no degradation in performance and at very low cost.

A more reliable and compact region delimitation hardware can be implemented if the values of the FAR and the CMD register are made accessible by the manufactures.

Chapter 6

Secure Computing Module

Contents

In this chapter, the structure for the Secure Computing Module (SCM) is proposed and its implementation on reconfigurable devices discussed. This structure is based on the Molen polymorphic processor, allowing to easily integrate the hardware implemented computational units with software code. This structure integrates the proposed cryptographic cores (Chapter 3 and 4), the attestation module (Chapter 5), and other mechanisms developed to realize the Secure Computing Module. Previous to the proposed Secure Computing computational structure, an overview of the Molen processing paradigm and of its design is presented in Section 6.1. Section 6.2 describes the proposed structure for the Secure Computing Module, including the hardware components and how to correctly use/integrate them in the programming code. The reconfiguration and programmability of the proposed structure, allow for the optional enforcement of the Secure Computing features, namely the Internal Attestation, the Secure I/O, the Sealed Storage, and the SCM unique identification (Endorsement Key). In section 6.3, the Cryptographic Computational Units (CrCUs) implemented to be used in the SCM are described, and their individual performance compared with related art. The full computational structure of the SCM is also compared with related cryptographic processors in Section 6.2. This performance comparison is made only in terms of ciphering throughput, disregarding the additional hardware mechanisms of the proposed SCM, since no related art exists for Secure Computing on reconfigurable devices. Section 6.5 concludes this chapter with an overview of the proposed Secure Computing structure.

6.1 Polymorphism and the Molen Paradigm

To implement the polymorphism functionality and to easily integrate the cryptographic hardware units into the SCM, the Molen paradigm [14, 74] is used. The Molen paradigm is based on the coprocessor architectural approach, allowing the usage of reconfigurable custom designed hardware units. In this computational approach, the non critical part of the software code is executed on a General Purpose Processor (GPP), while the critical and computationally more demanding functions/algorithms, are executed on the Custom Computing Unit (CCU). In the Molen paradigm, a hardware implemented function is seen by the programmer in the same manner as software implemented function. The decision where the function is executed is made at compile time. At microarchitectural level, the arbiter, depicted in Figure 6.1, redirects each instruction either to the GPP or to the CCU.

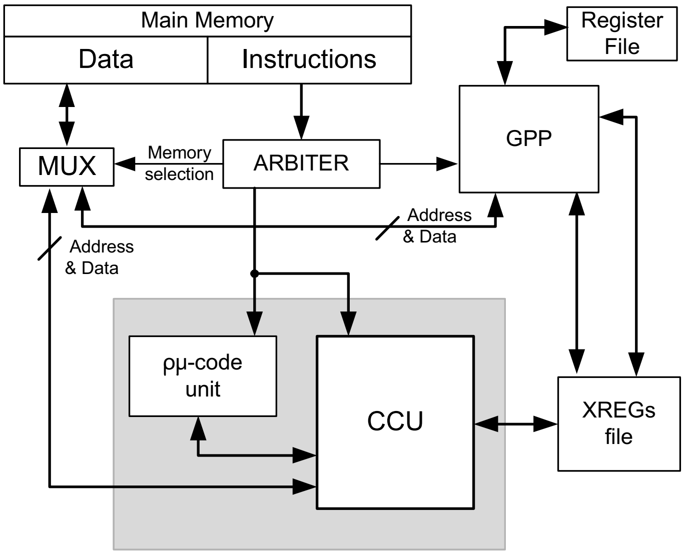


Figure 6.1: The Molen machine organization.

In a software function, parameters are passed to the function through the stack. In the Molen processor, when a hardware function is invoked the parameters are passed through a dedicated register file, designated by eXchange REGisters (XREG). To indicate which functions are executed in hardware, a $pragma$ annotation is used in the C code. This $pragma$ annotation is recognized by a dedicated compiler [14], which automatically generates the required set and $execute$ instruction sequence.

Given that the dedicated computational units are also connected to the main data memory, only initialization parameters are passed to the CCUs through the XREG. These parameters are memory pointers and initialization values, like the Initialization Vector (IV) or operation modes, such as indicating if encryption or decryption is to be performed. The data to be processed is directly retrieved or sent to the main data memory through shared memory.

In order to illustrate the data flow, the operations performed by the AES CCU are now described. When the AES cipher function is called, a few instructions are executed in the GPP, namely instructions that move the function parameters from the GPP internal registers to the XREG, followed by an $execute$ instruction. When an $execute$ instruction is detected by the arbiter, it starts addressing the microcode memory, gives the data memory control to the CCUs, and signals them to start the computation via the $start$ signal.

Once the AES core receives the *start* signal, it begging the retrieval of the values from the XREG. The first values read are the memory addresses where the AES key is stored. Continuously, while the start and end memory addresses for the data to cipher are retrieved from the XREG, the key is read from the (shared) main data memory. While the first data block is read from the memory, the initialization values are read from the XREG. After this initialization phase, the execution in the AES core enters a loop, where while the data is being ciphered the next 128 bits data block is read from the memory. At the end of each loop iteration, the ciphered data is written back into the data memory. When the current memory address coincides with the data end address, the computation loop is broken and the *stop* signal is sent to the arbiter. Upon receiving this *stop* signal, the arbiter returns the memory control to the GPP.

A *pragma* annotation is used in the C code, to indicate the location of the hardware implemented function in the execution path. Figure 6.2 depicts an example where the AES algorithm/function is executed in the CCU rather than in software. This *pragma* addition and recompilation are the only operations required to use the hardware instead of a software implemented function. With this mechanism, any execution path using such function is redirected to the CCU; the designer only needs to annotate the function declaration.

```
#pragma AES
AES (key, &data[0], &data[end], IV, mode){
   \* implemented in HW *\
}
```

Figure 6.2: Usage of the *pragma* notation.

In the Molen paradigm, the dynamic reconfiguration is performed by the *set* instruction, executed before the use of the CCU by the execute instruction. The *set* instruction specifies the configuration code that is to be uploaded into the device.

6.2 Secure Computing Structure

In this section, the computational structure of the implemented Secure Computing Module is presented and detailed. Figure 6.3 depicts the proposed structure, implemented on a Xilinx Virtex II PRO FPGA (XC2VP30-7) integrated on a Xilinx University Program (XUP) prototyping board. The used FPGA device includes two embedded PowerPC, one of which is used as the GPP of the

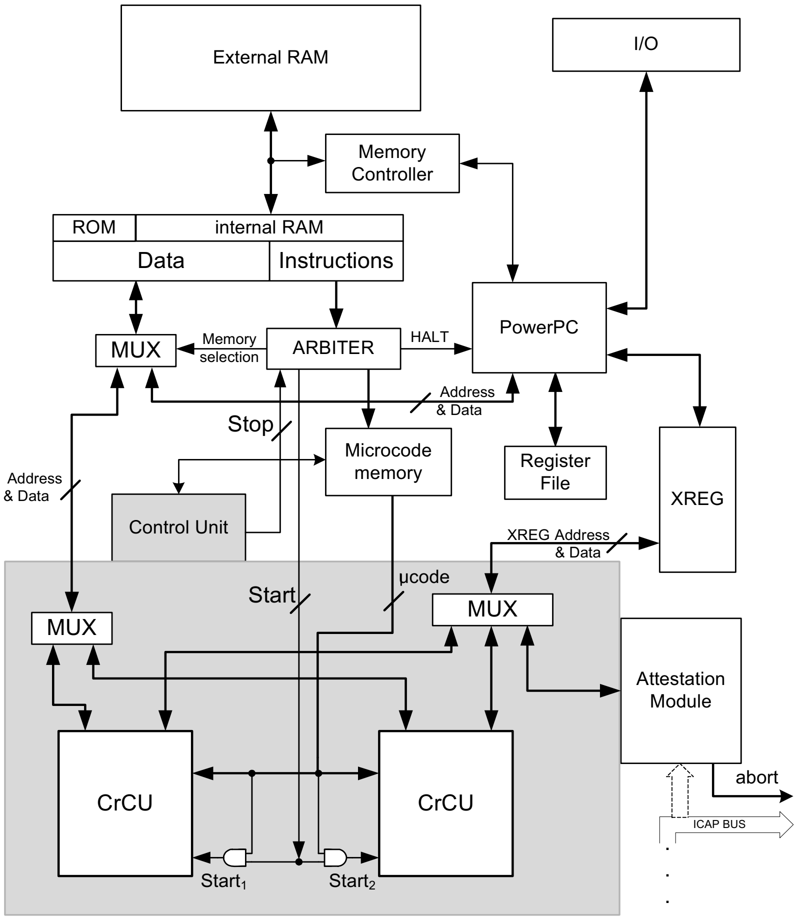


Figure 6.3: Secure Computing Module organization.

SCM. The XUP prototyping board includes: several I/O interfaces (e.g. RS-232, USB, and SATA), that can be used for the I/O interface of the SCM; external memory connectors for DDR SDRAM, to be used as the external memory of the module; and on board ROM, used only to store the initial configuration for the FPGA device. In order to exploit the functionalities made available by the Molen paradigm, the SCM was built based on the existing Molen processor [14]. Therefore, half of the device resources are used to implement the main GPP, the internal RAM, the microcode memory, the serial I/O interface, and the arbitration logic. Note that the majority of these resources are used for implementing the 128 kbytes of internal memory.

6.2.1 Multiple CrCU Allocation

Rather than having only one CrCU allocated at a given time, the proposed SCM allows the allocation of several simultaneous CrCUs. Taking into account that in most applications and protocols a set of algorithms is used, and that only this set of algorithms is used for a given temporal window, typically the duration of the connection to a server or the encryption of a data stream, the possibility of having several CrCU simultaneously allocated in the device becomes rather advantageous. Typically, during the establishment of a communication, the communicating parties agree on the cryptographic algorithms to be used. Since the reconfiguration times for such technologies are still significant [15], the constant permutation between the CrCU implementing the set of required algorithms would significantly degrade the SCM performance. The possibility of multiple allocated CrCUs, that implement the algorithms agreed to be used for the communication, allows to disregard the CrCUs reconfiguration time. These units are uploaded to the device during the protocol agreement phase, causing no performance degradation during the data communication phase.

In Figure 6.3, a SCM structure with 2 allocatable CrCU is depicted. When the $execute$ instruction is executed in the GPP, an address is sent to the microcode memory. This microcode is then used to select which CrCU is to be used, namely by generating a $start$ signal only to the intended unit. This value is also used in the selection logic depicted in the grey area of Figure 6.3.

To properly manage the space available for the CrCUs, the available reconfigurable area resources are partitioned into slots. Each slot is reserved for the allocation of one CrCU within the SCM; each CrCU can be allocated on an empty slot, independently of the other already allocated CrCUs. If a given CrCU requires more reconfigurable hardware area than that reserved in each slot, several contiguous slots can be used to allocate it. The number of available CrCUs slots is predefined in each version of the SCM. In this thesis, versions of the SCM with 1, 2, and 4 CrCUs were implemented to obtain the experimental results. If sufficient slots are not available, the required CrCUs will need to be frequently reallocated, thus degrading the global performance of the SCM (see Section 6.3).

Unlike the typical Molen usage, in the proposed Secure Computing Module the uploading of a new CrCU cannot be simply done with the set instruction introduced by the compiler, before the $execute$ instruction. Since several CrCUs can be allocated in the device at the same time, the space available for the CrCUs allocation has to be properly managed. Also, the uploading of the CrCU into the FPGA has to be certified, to attest for the hardware validity.

6.2.2 Attestation on the Secure Computing Module

In order to assure the correct operation of the SCM, the reconfigurable hardware has to be verified before its usage. One approach would be to view the hardware configuration file as a set of software instructions or data that has to be validated, i.e. loaded to the internal memory and verified before being used. However, the dimension of the configuration files tend to be significantly large, implying a long computational time to: (i) load the bitstream data into the internal memory; (ii) generate the hash value for the whole data file; (iii) and finally, send the data to the FPGA configuration bus to load the CrCU. The other, significantly more efficient, alternative is to perform the bitstream attestation on the fly. However, the DS of the bitstream can only be verified after the DM of all the data is generated. Therefore, if a corrupted or invalid bitstream is used, any part of the SCM could be reconfigured, including the attestation module itself. An attacker could target the hardware responsible for the generation of the DM, putting whatever value he would desire. To prevent this type of attacks, the attestation module includes a write boundary checker: if an attempt is made to write in a region other than the CrCU slot region, an abort signal is generated and the hardware reconfiguration stopped, preventing any irreversible corruption of the SCM.

When an invalid bit stream is detected, all available slots have to be marked as invalid. This is needed due to the fact that an incorrect configuration data file may write into other slots than that for which it was meant, corrupting them. The attestation module only stops an ongoing reconfiguration, in the case of an attempt to write outside the CrCUs area boundary delimitation.

To assure that the loaded CrCU is in fact the correct one, some software support is required. The attestation module by itself assures that no data is written outside the boundaries of the reserved CrCU space. However, it does not assure that the loaded CrCU is in fact correct; it simply generates the DM of the loaded CrCU. To assure that the loaded CrCU is in fact the correct one, the DM generated by the attestation module has to be compared with the original DM of the loaded CrCU. The original DM values of the allowed CrCUs for a given SCM are stored in a table designated by validation table. This table can be locally stored on the SCM ROM or as constant values in the initialization code of the SCM. However, due to the physical limitation of the device and its efficient usage, the internal ROM may not allow the storage of all digital signatures for all of the execution path sections (or bitstreams). An alternative solution is to only store the DM of the validation table root, in the secure internal ROM. Figure 6.4 depicts a block diagram of a two level scheme for such tables, where the root value is the value stored in the ROM; this value is the DM of the table with the Digital Signatures, herein designated by validation table. In order to validate an execution path, its DM is generated and compared with the DM obtained from the validation table. If it matches, only the validation table itself has to be attested; this verification is required since this table might be stored in an unsecured memory region. The validation of the second level of the validation table is accomplished by generating its DM, and comparing it with the entry of the DM in the first level validation table. Finally, the DM of the first level validation table is compared with the root DM located in the secure memory (ROM). If all these tests are valid, then the tested execution path can be considered secure. Given that a SC approach is being used, and that only the internal attestation is being considered, the DM of the executed path is sufficient for the internal attestation. In the implemented prototype, a validation table embedded in the bitstream is used.

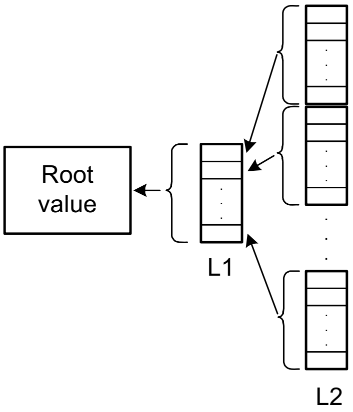


Figure 6.4: Digital Signatures table hierarchy.

Figure 6.5 depicts the pseudo code used to load the AES CrCU in a secure manner. First, it is checked if the CrCU has already been loaded; if this is the case, nothing else is done. Assuming that the AES CrCU has not yet been loaded, the set instruction is invoked, in order to load the AES CrCU into the FPGA. Once the CrCU has been loaded, the attestation module is read. In the $compare()$ function, the DM of the loaded CrCU is compared with the value stored in the validation table. The $compare()$ function also tests the *Abort value*, to determine if the configuration bitstream tried to write outside the CrCU allocation space. If the valid AES CrCU was properly allocated, the $current\_CrCU$ variable is updated with the value indicating that the AES

CrCU is loaded, and a $valid$ flag is returned. In case an invalid CrCU has been uploaded, the current CrCU variable is set to $empty$ and the $invalid\_CrCU$ exception is called. Note that in the proposed SCM, the set instruction is explicitly introduced, rather than implicitly by the compiler, as is the case of the Molen processor [14].

```
Load_CrCU( ) {
if (not (Current_CrCU == AES)) then
    set(AES)
    DM = read_DM( )
    if ( compare( DM , read_VT(AES) == not_equal) then
        Current_CrCU = NULL;
        invalid_CrCU( );
    end if
    Current_CrCU = AES ;
end if
return (valid);
}
```

Figure 6.5: Loading of a CrCU into the SCM.

With this attestation module and software validation procedure, the Internal Attestation feature for the Secure Computing hardware can be enforced. With the attestation of the CrCUs being executed in hardware and in a separate module, parallelism between the CrCU allocation process and computation on the SCM, not requiring this unit, can be explored. For example, the allocation/reconfiguration of a CrCU may be occurring while part of the communication protocol is still being agreed upon the SCM and other communication entities. Additionally, since the CrCU is attested while it is received, the bitstream data can be retrieved directly from an external unsecured memory or repository, while the SCM internal memory is being used in some other computation.

6.2.3 Remaining SCM Structure

As stated in the beginning of this section, half of the FPGA is used in the implementation of the internal RAM memory of the Secure Computing Module. This memory is implemented with the embedded RAM blocks of the FPGA. Half of this memory is used to store the instructions executed on the GPP, while the other half is used as data memory.

Although a ROM memory exists on the XUP board, this ROM can only be used to store the FPGA initialization bitstream. Moreover, since the ROM is located outside the FPGA device, the implemented SCM is susceptible to physical attacks, e.g. tampering with the bitstream, during the device initialization, when the bitstream is being transmitted from the ROM to the FPGA. The implementation of the SCM on an FPGA with internal non-volatile memory, will allow a significantly higher resistance to this type of physical attacks. The weakness of having the configuration ROM located outside the chip can be minimized, by storing the initialization bitstream encrypted and only decrypting it inside the FPGA device. Current devices, however, do not allow the use of partial dynamic reconfiguration when encrypted bitstreams are used [15, 73].

The internal data ROM will also become available, to store critical data, like encryption keys. By storing the Endorsement Key(s) on the internal secure ROM, the SCM can be uniquely identified, since this key or keys are only known inside the module. In the implemented version of the SCM, the key is hardwired into the configuration bitstream. The existence of several Endorsement Keys allows the creation of multiple virtual Secure Computing Modules, adding extra privacy to the users. The flexibility of this module allows the option to give to the user write access to the memory region where the Endorsement keys are stored, granting him permission to change the modules Endorsement keys.

Regarding the usage of the RAM, all data is first written into the internal memory, protected by the physical security of the device. The data can first be ciphered before it is written into the external memory. If the data is only found deciphered in the internal memory of the SCM, the Sealed Storage feature of the Secure Computing is enforced.

Identically, the I/O data is first stored in the internal memory, where it can be ciphered before being written into an external unprotected environment, thus enforcing the Secure I/O feature of Secure Computing.

6.3 Cryptographic Computational Units

This section presents the utilization of the cryptographic cores, proposed in Chapters 3 and 4, as Cryptographic Computational Units (CrCU) integrated in the SCM. Each CrCU has been individually integrated and their performance evaluated. The resulting version of the single CrCU SCM was compared with related cryptographic processors. The obtained results in this evaluation do not consider the reconfiguration or the attestation costs. The experimental results presented in this section were obtained for a Xilinx Virtex II PRO-7 FPGA on a XUP prototyping board. All results presented in this section were obtained for an operation frequency of 100MHz, not the maximum frequency of each individual CrCU; this frequency was used in order to create prototypes with the same operating conditions. This option also facilitated the integration of the CrCUs, given that 100MHz is the internal memory operating frequency.

For the two implemented symmetrical algorithms CrCU (AES and DES), the computation flow is the one described in Section 6.1. For the hashing algorithms (SHA and Whirlpool), the data flow is identical, with the particularity that a hash function is a compression function that only generates data at the end of the entire computation; thus, during the computational loop no data is written into the memory. Once all data blocks have been computed, the final Digest Message is written into the memory, and the computation concluded with the activation of the $stop$ signal.

The following presents the performance analysis for the AES, DES, SHA128, SHA2, and Whirlpool Cryptographic Computational Units. The PowerPC embedded in the FPGA is used as the GPP running at 300MHz, and with the internal RAM memory of the SCM operating at 100MHz.

6.3.1 AES CrCU

On a Xilinx Virtex II PRO-7 FPGA, the AES core proposed in Chapter 3, is capable of a throughput of 2.3Gbit/s at a maximum frequency of 180MHz, requiring 515 Slices and 12 BRAMs, 4 of which are used to store the expanded key. In order to create the AES CrCU additional glue and control logic was added to the AES core. The proposed AES CrCU was implemented with a fully folded loop AES core, able to encrypt and decrypt in both ECB and CBC modes. The complete AES-CrCU requires 1130 Slices and 12 BRAMs. In this implementation, the key expansion is performed in software.

The implemented AES CrCU supports all key sizes, i.e. ciphers the data for 10, 12, or 14 rounds. The operation modes as well as the IV for the CBC mode are passed as additional parameters through the XREG. In order to standardize the results, only the throughput results for the ECB mode are presented at a frequency of 100MHz. Table 6.1 presents the throughput results using a 128-bit key, as well as the speed-up obtained with the AES core, regarding a software implementation.

The obtained speed-up is just 43 when only one 128-bit data block is encrypted, due to the overhead of transferring the expanded key (1408 bits). Note however, that the overhead of the expanded key transfer becomes negligible when considering the encryption of large files; for a data stream of 16kBytes

Table 6.1: AES CrCU performances @100MHz

	Hardware		Software		
Bits	Cycles	(Mbit/s) ThrPut	Cycles	(Mbit/s) ThrPut	Kernel SpeedUp
128	646	59	24216	1.59	43
4k	4366	281	738952	1.66	169
128k	31246	1258	23610504	1.67	751

a speed-up of 571x is accomplished. The last column in Table 6.1, contains the calculated speed-up for the ciphering kernels. The number of cycles is the same for encryption and decryption, in both the software and hardware implementations. Note that these speed-up values only consider the ciphering subroutine (kernel) and not the entire application. In practical applications, even if only one data block is ciphered in every function call, the expanded key remains the same. Since the AES core internal registers are not cleared each time a new encryption is executed, the expanded key only has to be transferred once. In this case, for the ciphering of a single 128-bit data block per function call, a hardware throughput of 89Mbit/s is obtained, resulting in a local kernel speed-up of 56. Typically, the data blocks being ciphered tend to be bigger than 16 kBytes; in these cases a speed-up above 750 can be achieved.

The SCM running the AES CrCU is compared to a recently published, computationally identical processor [75]. The figures presented in Table 6.2 show an improvement on the required area, while maintaining an identical throughput for the 128-bits key cipher mode. For the ciphering, using 256-bit keys, the implementation proposed in [75], the operating frequency significantly decreases; the proposed AES-CrCU is capable of cipher with all key sizes at the same frequency. For the AES ciphering with a 256-bit key, the proposed SCM achieves a 16% higher throughput. The AES version for 256-bit keys in [75] also implies an area increase, regarding their 128-bit AES core, thus reducing the Throughput per Slice performance metric.

The Throughput per Slice ratio given in Table 6.2 relates to the most advantageous case for the processor proposed in [75]. Even in this case, this metric is 59% higher. For the 256-bit key ciphering, this Throughput per Slice improvement is even higher.

---

[1] A estimated value for the slice utilization has been used, for a ratio of 0.62 Slices per LUT. The authors in [75] only give the number of used LUTs. The proposed AES CrCU has a 0.62 Slice/LUT ratio.

Table 6.2: AES processors

Architecture	Lu [75]	AES CrCU
Cipher	Enc./Dec.	Enc./Dec.
Operation modes	ECB, CBC	ECB, CBC
Device used	XC2VP100	XC2VP20
Main processor	PowerPC	PowerPC
Number of slices	1700[1]	1130
Number of BRAM	44	12
Max. frequency (MHz)	196-179	100
ThrPut AES-128 (Mbit/s)	1197	1258
ThrPut AES-256 (Mbit/s)	778	905
TP/S	0.7	1.1

6.3.2 DES CrCU

In order to maintain the backward compatibility with older applications, a DES CrCU has been included into the SCM. The DES CrCU is implemented with a straightforward structure, using the embedded memories of the FPGA. This DES core only computes the single DES encryption and decryption, and in ECB mode. For the computation of 3DES algorithm, the DES CrCU is invoked 3 times with the appropriate keys.

On a Xilinx Virtex II PRO-7 device, the standalone version of the DES core is capable of a throughput of 872Gbit/s at a maximum frequency of 218MHz, requiring 175 Slices. With the control and glue logic, the DES CrCU requires 328 Slices and 4 BRAMs.

Comparative results for pure software and hardware implementations are presented in Table 6.3. This table also presents the speed-up achieved for the kernel computation of the DES algorithm. Once more, a difference in the ciphering throughput can be seen, for different block sizes. This is due to the initialization overhead cost of the DES CrCU, which includes the loading of the key and the transfer of the data addresses from the XREG to the DES CrCU.

Table 6.3: DES CrCU performances @100MHz

Bits	Hardware ThrPut	Software ThrPut	Kernel SpeedUp
64	89 Mbit/s	0.92 Mbit/s	97
128	145 Mbit/s	1.25 Mbit/s	116
4k	381 Mbit/s	1.92 Mbit/s	198
64k	399 Mbit/s	1.95 Mbit/s	205

This initialization overhead becomes less significant as the amount of data to be ciphered increases, becoming negligible for data blocks above 4 kbits. A speed-up of about 200 can be obtained, achieving a ciphering throughput of 399Mbit/s at a frequency of 100 MHz.

Table 6.4 presents the figures for the DES CrCU and for related DES art [76]. It can be seen that the proposed SCM running the DES CrCU is able to outperform the related art in terms of throughput by 30% with less than 40% of FPGA occupation. These results suggest a Throughput per Slice improvement of 117%.

Table 6.4: DES processors

	Chodo [76]	Our-LUT	Our-BRAM
Device	V1000	V1000E	V2P30-7
Freq. (MHz)	57	100	100
FPGA usage	5%	3%	2%
ThrPut - DES (Mbit/s)	306	399	399
ThrPut - 3DES (Mbit/s)	102	133	133

6.3.3 SHA CrCU

On a Virtex II PRO-7, the proposed SHA-1 core [63] is capable of a compression throughput of 1.4Gbit/s at a maximum frequency of 230MHz, with 565 Slices, while the SHA-2 core [64] (256 bits) is capable of a compression throughput of 1.4Gbit/s at a maximum frequency of 170MHz, with 755 FPGA slices and 1 BRAM. When implemented as CrCUs, the SHA-1 unit requires 813 Slices and the SHA-2 unit requires 994 Slices and 1 BRAM. Both cores allow the loading of a given IV, facilitating the computation of HMAC and the computation of fragmented messages.

SHA-1 CrCU

Table 6.5 compares purely software and hardware implementations of the SHA-1 hash function.

Even though the SHA-1 algorithm can be efficiently implemented in software, achieving a throughput above 4Mbit/s, the usage of the SHA-1 CrCU allows for a speedup up to 150. For data streams with only a few data blocks, a lower speed-up is obtained, once again due to the CrCU initialization overhead. Even so, a speed-up of approximately 100 times is still achieved for the worst case

Table 6.5: SHA-1 CrCU performances @100MHz

	Hardware		Software		
Bits	Cycles	(Mbit/s) ThrPut	Cycles	(Mbit/s) ThrPut	Kernel SpeedUp
512	396	389	38280	4.01	97
1024	642	479	76308	4.03	119
128k	63126	623	9766128	4.03	155

usage. For data streams with several data blocks, the achieved speed-up rapidly tends to 150 times.

The proposed SHA-1, used in the SCM, is compared with the related art, reported in [75]. As depicted in Table 6.6, the proposed implementation is able to achieve a 100% higher throughput with significantly less hardware resources; therefore, a 670% higher Throughput per Slice is obtained.

Table 6.6: SHA-1 processors

Design	Lu [75]	Our+IV
Device	XCV2P100	XCV2P30-7
Slices	3441[2]	785
Freq. (MHz)	145	100
ThrPut (Mbit/s)	304	624
TP/S	**0.1**	**0.77**

SHA-2 CrCU

Identically to the SHA-1 CrCU, SHA-2 CrCUs have also been realized. The CrCU for the SHA256 core requires 994 Slices using in total 7% of the available hardware resources of the Virtex II Pro 30 FPGA. The CrCU for the SHA512 core requires 1806 Slices using in total 13% of the available hardware resources. Table 6.7 presents the speed-up achieved with the use of the SHA256 hardware, compared with the pure software implementation of the SHA256 algorithm. The obtained figures suggest a speedup up to 153. This speed-up is achieved when the total data size is sufficiently large to compensate the initialization of the CrCU, corresponding to a throughput of 785Mbit/s. When only one data block is hashed, the initialization time becomes more relevant, reducing the speed-up to 85. When at least two data blocks are sent,

---

[2] Synthesis results for the SHA-1 core only. An estimated value for the slice utilization has been used, for a ratio of 0.58 Slices per LUT, obtained in the proposed SHA-1 CrCU.

Table 6.7: SHA256 CrCU performances @100MHz

	Hardware		Software		
Bits	Cycles	(Mbit/s) ThrPut	Cycles	(Mbit/s) ThrPut	Kernel SpeedUp
512	354	434	30402	5.05	85
1024	552	556	60546	5.07	109
128k	50088	785	7718646	5.09	153

the initialization becomes less significant; a speed-up of 109 is achieved. The implemented SHA512 CrCU is capable of achieving a maximum throughput of 1.2Gbit/s.

6.3.4 Whirlpool CrCU

In order to use the proposed Whirlpool core as a CrCU unit in the SCM, some additional control and glue logic was added. This additional logic does not penalize the critical path of the unit. The Whirlpool CrCU requires 2245 Slices (16%) and 32 BRAMs, which correspond to 127 additional slices regarding the standalone Whirlpool core. Table 6.8 presents the throughput values for different package sizes. When only a 512-bit package is sent, a throughput of 545Mbit/s is achieved. When bigger packages are sent, the initialization of the CCU become less significant; a throughput of 2.4Gbit/s is achieved, for a package with at least 16kBytes.

Table 6.8: Whirlpool CrCU performance @100MHz

Bits	Cycles	(Mbit/s) ThrPut
512	94	545 Mbit/s
1024	114	898 Mbit/s
128k	5448	2.4 Gbit/s

6.4 SCM Evaluation and Related Art

In this section, the SCM is evaluated and compared with related art. To the best of the authors' knowledge, no other security related processors are capable of attesting dynamically reconfigured hardware. The compared art are processors and co-processors implemented on reconfigurable devices; the compared art are statically reconfigured.

Table 6.9 summarizes the characteristics of the proposed CrCUs, regarding the Virtex II Pro 30 implementation, a midrange FPGA from Xilinx.

Table 6.9: CrCUs on a V2P30-7 @100MHz

	DES	AES	SHA-1	SHA256	SHA512	Whirlpool
ThrPut (Mbit/s)	399	1258	624	785	1200	2400
Slices	3%	4%	6%	7%	13%	16%
BRAMs	3%	9%	0%	1%	1%	16%

It can be seen that, individually, each CrCU requires less than 15% of the total available resources. The Whirlpool CrCU is the exception, requiring about 16% of the available hardware sources. It should be taken into account that, this CrCU has been developed to achieve the maximum performance; if less than 16% of the available resources are available for this CrCU, some hardware reuse techniques can be applied to the Whirlpool core: less than 10% occupation of the FPGA is expected for a throughput of 1.2Gbit/s. An identical procedure can be applied to the SHA512 CrCU. In conclusion, each CrCU can be made to occupy less than 10% of the available resources, allowing the simultaneous use of several CrCU on the device. As described in Section 6.2, half of the device is used to implement the SCM, including the GPP, the internal memory, and the arbitration logic. Therefore, 50% is free for the implementation of the attestation module and the CrCUs. With the attestation module, described in the previous Chapter, using 10% of the available hardware resources, 40% remain available for the CrCU.

Table 6.10 describes the resources required when 2 CrCU are simultaneously allocated on the SCM. These values include the multiplexing and section logic, depicted in the grey area of Figure 6.3.

Table 6.10: Multiple CrCUs occupation values on a V2P30

	DES-SHA1	AES-SHA1	DES-SHA2	AES-SHA2	4 CrCU
Slices	1668	2094	1947	2240	4022
Usage	12%	15%	14%	16%	29%
BRAMs	1%	9%	2%	9%	10%

The figures in Table 6.10 suggest that the use of 2 CrCU correspond an average occupation of 15% of a Xilinx Virtex II Pro 30, approximately 2000 Slices and a maximum of 13 BRAMs. Using 512-bit hashing CrCUs, the average occupation increases to approximately 20% of the device.

These figures were obtained to have an idea of the SCM average occupation. In the practical utilization of the SCM, the CrCU allocation has to be made modular. In the used prototype SCM, the available 40% of the FPGA are divided into 2 rectangular slots with 20% of the available area resources. These 20% take into account the worst case scenario for CrCU occupation, namely the 16% occupation of the Whirlpool CrCU. If a particular CrCU requires more than 20%, the 2 slots can be used to allocate this CrCU. A more fine-grained organization for the space available for the CrCUs can be arranged, for example 4 slots with 10% of the available resources. I this case, the SHA512 and the Whirlpool CrCU would require 2 adjunct slots of 10%. It is worth noticing that, a finer grained space organization will require more complex allocation algorithms.

A version of the proposed SCM with 4 CrCUs (AES, DES, SHA-1, and SHA256) has also been implemented, in order to extrapolate the cost of not having reconfigurability, or when more CrCUs need to be simultaneously allocated in the SCM. While the same throughput is achieved, as the SCM with 2 CrCUs, the area requirements increase to approximately 100% more, i.e. 4000 Slices, representing 30% of the total available resources. The optimal balance between the space reserved for each CrCU and the maximum number of CrCU that can be simultaneously operating, depends on the target applications. The configuration of 10% of the Virtex II Pro 30 device requires approximately 3ms, while the encryption of a 128 kbit data stream requires less than 0.3ms for the slowest CrCU. This means that the performance of the SCM can be significantly affected if the CrCUs have to be constantly allocated, more than 10 times in the above example. As the number of CrCU increase, so does the selection data path, which may cause degradation of the performance. This is not the case in this experimental setup, since a conservative frequency of only 100MHz is used in the prototype. A prototype operating at higher frequencies would allow for better performances, since the CrCUs maximum operating frequencies are not limited to 100MHz.

Related art

Since the existing art does not perform any sort of dynamic reconfiguration or attestation, the area cost of the Secure Computing units like the attestation module and the dynamic reconfiguration are not considered in the performance evaluation, regarding the other cryptographic processors. For the same reason, 512-bit hashing units (the SHA512 and the Whirlpool CrCU) are not considered, since the related art only uses hashing functions up to 256 bits.

To properly compare the proposed processor with the related art, implementation results for a Xilinx Virtex 1000E FPGA have also been obtained. Note that for the Virtex 1000E reconfigurable technology, the CrCUs are unable to achieve an operating frequency of 100MHz, especially the AES CrCU (the used AES core has a memory based structure optimized for 32-bit memories). The 16-bit memory structure of the Virtex-E family originates a less efficient AES CrCU unit, only capable of achieving a maximum operating frequency of 55MHz. It is worth to notice that our design has not been optimized for the Virtex-E technology.

In Table 6.11, the figures of the proposed SCM are compared with the related cryptographic processors. In this table, the algorithms that are not available in a given processor are marked as *not available* (n.a.). The required hardware resources for the SCM are presented with $\leq$, indicating the maximum occupation values for a set of two CrCUs.

Table 6.11: Cryptographic Implementations

	McLoone [61]	Chodo [76]	SCM	Lu [75]	SCM
FPGA	XCV1000E	XCV1000	XCV1000E	XV2P100	XV2P30-7
Slices	7247	~ 1800	≤ 2000	~ 8300	≤ 2240
BRAMs	20	18	≤ 40	~ 44	≤ 13
Freq. (MHz)	24	57	$55 - 78$	$107 - 145$	100
	Throughput (Mbit/s)				
AES	310	404	699	$653 - 886$	1258
DES	n.a.	306	$222 - 350$	n.a.	399
3DES	n.a.	102	$74 - 117$	n.a.	133
MD5	n.a.	n.a.	n.a.	277	n.a.
SHA-1	$38 - 78$	n.a.	$347 - 589$	304	624
SHA-2	n.a.	n.a.	$436 - 688$	n.a.	785

When compared with the proposal by McLoone [61], implemented on a Virtex-E, an occupation reduction of 30% is achieved, with a speed-up of more than 2 times for AES and more than 7 times for SHA-1. The improvement is not so significant when compared with the Chodo [76] proposal: the area results are slightly higher in the proposed SCM; however, the AES computation is accelerated by 70%.

When compared with Lu's [75] IPSEC implementation on a Virtex II PRO, the SCM achieves an area reduction of 270% and accelerates the AES and SHA computation by approximately 2 times. A Throughput per Slice improvement of more than 350% is obtained, even with the proposed SCM operating at 100MHz. Taking into account that the Lu [75] IPSEC implements 3 algorithms, the SCM with 4 CrCU has also been used in the comparison; in this case an area reduction of 100% is still achieved.

Experimental testing

In order to apply the proposed Secure Computing Module in a real functional environment, the module was connected to a PC running Linux through the serial port. For this functional test, several files, e.g. jpeg images, were sent from the PC's hard drive to the SCM, encrypted or decrypted using the AES algorithm, and sent back to the PC. The bottleneck is located on the image transmission, since a serial com port at 19.2kbps is used in this test. Consequently, no performance measures were done; the data transmission time (19kbps) is significantly larger than the ciphering time (of 1Gbps). The tests show the correct functionality of the SCM, when the ciphering keys were either internally stored within the SCM secure memory or transmitted via the I/O. A more efficient utilization of the SCM can be made when faster I/O interfaces are used, e.g. LAN or USB.

6.5 Conclusions

In this Chapter, the complete structure for the Secure Computing Module is proposed. With the use of the Molen paradigm the integration of different Cryptographic Computational Units (CrCUs), into the Secure Computing Module, can be made with minimal effort in the software code, and high ciphering throughputs are achieved. The proposed SC structure allows the Secure Computing features to be properly implemented while maintaining an adequate performance for the system. The use of the internal memory within the physical security of the device, along with the fact that all the data has to go through this memory, and the use of efficient CrCUs allow for the implementation of the Secure I/O and Sealed Storage features of Secure Computing. The use of the attestation module allows for the dynamically reconfigurable hardware of the SCM, in particular the CrCUs, to be loaded and used in a trustworthy manner. The use of delimited slots allows for a safer and easier integration of new cryptographic cores. The multiple CrCU allocation organization, allows the masking of the reconfiguration time when several algorithms are used in the same temporal windows. Since only the required hardware functions are uploaded to the reconfigurable device, a lower reconfigurable area is required. Experimental results validate the performance of the proposed Secure Computing Module. With an average occupation of 2000 Slices (15%) when using 2 CrCU, throughputs above 1Gbit/s are achieved, resulting in speedups ranging between 150 to 750 times, regarding pure software implementations of the cryptographic functions. When compared to related cryptographic processors, the proposed Secure Computing Module is more adaptable and capable of achieving a higher performance with less reconfigurable hardware area. Implementation results suggest Throughput per Slice improvements from 100% to 350%.

Chapter 7

Conclusions

Contents

Secure computing (SC) is becoming a mandatory requirement in the current age of information and digital processing. The industry recently established a set of features used to achieve the required level of security and trustworthiness over the used computational systems. An industrial consortium has been established to develop the Trusted Platform Module (TPM) chip. This chip, however, is based on a well defined static system, lacking the adaptability required to be employed in more customizable and reconfigurable computing environment. This thesis proposes an equivalent module to the TPM but, capable of being used in a wider range of systems and applications, and of being able to rapidly adapt to newer demands and threats.

This chapter presents an overview of the work developed towards a Secure Computing hardware platform based on reconfigurable devices. Section 7.1 summarizes the work and results presented in this thesis regarding the Secure Computing Module (SCM). Section 7.2 highlights the main contributions of the thesis to design the proposed hardware structures to support for Secure Computing. Concluding this thesis, future research directions are presented in Section 7.3.

7.1 Summary

In this dissertation a Secure Computing Module on reconfigurable systems is proposed. The proposed module, allows for the Secure Computing (SC) features to be properly implemented on reconfigurable devices while maintaining an adequate performance. By allowing different sets of Cryptographic Computational Units (CrCUs) to be dynamically allocated to the SCM, the computational structure can be adapted to the desired application or system. Given that the system designer is capable of changing the execution flow in the SCM, he has the ability to use only the features required. For example, if the FPGA configuration bitstreams for the CrCUs are stored in an external memory that the system designer deems to be secure, the attestation module can be deactivated, or he may simply use the attestation module to assure region delimitation, avoiding the integrity test for the loaded CrCU.

The use of an attestation module allows for the dynamic reconfigurable hardware of the SCM, in particular the CrCUs, to be loaded and used in a trustworthy manner. With the attestation of the configuration bitstream in hardware, the reconfigurable hardware structures can be loaded into the device with practically no performance degradation at reduced area cost. The secure initialization of the SCM, along with the attestation module allow for the Internal

Attestation feature to be assured.

Regarding the Endorsement Key required for the Secure Computing, the value or values can be either hardwired into the configuration bitstream, if an internal ROM is not available for the SCM; or alternatively, stored in the internal non-volatile memory of the SCM. By allowing the use of multiple Endorsement Keys and the optional write access to the user, multiple virtual Secure Computing Modules can exist, giving to the user of the SCM an additional privacy capability. Remote attestation to the computation performed in the reconfigurable SCM cannot be guaranteed to an external entity, since a user has control over the system and how it behaves.

The use of the internal memory within the physical security of the device allows the implementation of the Secure I/O and Sealed Storage features of Secure Computing. This feature can be efficiently used, given the existence of high performance CrCUs to compute the required symmetric encryption algorithms.

Typically, secure applications and protocols only require a reduced set of algorithms during a given secure session, which can be defined during the initial protocol agreement. By associating this characteristic with the proposed multiple CrCU organization, the performance penalty imposed by the reconfiguration time can be disregarded, since the CrCUs do not have to be reallocated during computation. The dynamic partial reconfiguration capability and the uniform structure proposed to integrate the several CrCUs on a reconfigurable device, allow for a wide variety of algorithms to be used, with modest reconfigurable hardware resources. These features also provide a high adaptability of the SCM to use new protocols and safer CrCU implementations, for example hardware structures resistant to new forms of Differential Power Attacks (DPA) [77,78].

The most significant results and the main conclusions of each chapter of this dissertation are summarized below.

In chapter 3, computational structures for the two most relevant symmetrical encryption algorithms, DES and AES, are proposed. The availability of a DES core is important to maintain backward compatibility. The proposed structure is capable of achieving a throughput of 1Gbit/s, and suggests that the use of BRAM to implement the substitution boxes allows the improvement of the Throughput per Slice metric by 20%.

Regarding the most relevant symmetrical encryption algorithm in current applications, the AES algorithm, a compact structure with high throughput is achieved. By merging the two most demanding operations of the AES algorithm, the *i)* byte substitution and GF(2^8) multiplication, into a single BRAM, several improvements are achieved, namely: the encryption and decryption ciphering can be efficiently performed using the same hardware; *ii)* a significant reduction in the required amount of FPGA slices; *iii)* a smaller critical path, leading to a high ciphering throughput. The proposed AES core achieves a maximum throughput 34% faster than any known existing AES core, and requires 68% less reconfigurable hardware, suggesting an improvement on the Throughput per Slice metric of more than 200% when compared to the state-of-the-art research and leading commercial products. The folded AES core is capable of achieving a throughput above 2Gbit/s. For the unfolded structures, the Throughput per Slice metric is improved 560%, achieving a throughput above 34Gbit/s.

In chapter 4, the three hashing algorithms most likely to be used in the future are presented, namely the SHA-1, SHA-2, and Whirlpool algorithms. Regarding the SHA algorithms, the proposed hardware rescheduling and reutilization techniques improve the hardware realizations, suggesting improvements to the Throughput per Slice metric between 29% to 100% when compared with commercial products and current academia art. Compression throughputs above 1Gbit/s are achieved with reduced hardware resources requirements.

A computational structure for the Whirlpool hash function is also proposed. Identically to the AES structure, the S-Boxes and part of the GF(2^8) multiplications are merged into a single operation and performed through a lookup table implemented in a single BRAM. Since the most complex part of the GF(2^8) multiplication is computed by a lookup table, a faster and more compact structure is achieved. Moreover, the merging of the round key computation and the data compression hardware is also proposed, resulting in an improvement of the Throughput per Slice to related art of about 160%. A throughput of 5.5Gbit/s is achieved, with only 2110 Slices and 32 BRAMs.

In chapter 5, a novel attestation method is proposed to validate the hardware structures loaded into the reconfigurable device in run time. The attestation is performed by computing in hardware the Digest Message of the bitstream and comparing it against the expected value. An additional region delimitation mechanism is also used to assure that an adulterated bitstream is not able to damage or modify the Secure Computing Module itself. With the region delimitation mechanism, an attempt to modify any resources outside the delimited area, leads to the halting of the reconfiguration process. With the proposed attestation module, assurance in the dynamic reconfiguration of hardware structures can be accomplished with practically no degradation in performance and with low hardware costs.

Chapter 6 concludes this research work, combining the proposed hardware structures to form the Secure Computing Module. Experimental results validate the performance of the proposed Secure Computing Module. With an average occupation of 2000 Slices (15%) of the used Virtex II Pro 30 FPGA when allocating 2 CrCUs, throughputs above 1Gbit/s are achieved, with speedups ranging between 150 to 750 times, regarding pure software implementations of the cryptographic functions. When compared to related cryptographic processors, the proposed SCM is more adaptable and capable of achieving an improved performance with less reconfigurable hardware area. Implementation results suggest Throughput per Slice improvements ranging from 100% to 350%.

7.2 Contributions

In this section the main contribution of this dissertation are summarized. These contributions are organized in two main components. One related with the design of Secure Computing Module on reconfigurable systems and the other with proposed ciphering hardware structures.

Secure Computing Module for reconfigurable systems:

- A novel attestation module capable of run time attestation of dynamically reconfigured hardware structures is proposed. With the use of region delimitation hardware, and the hash computation of the configuration bitstream in hardware, the attestation of reconfigurable hardware can be accomplished without performance degradation and at a low reconfigurable area cost.

- The design of high performance CrCUs, resulting in speedups ranging between 150 to 750 times, regarding pure software implementations. Improvements to related cryptographic processors up to 350% are also suggested.

- As the main goal of this dissertation, a secure computing module was proposed implemented on a reconfigurable system. The Secure Computing prototype is capable of throughput above 1Gbit/s while accommodating the features required for creating a secure computing environment.

High performance ciphering hardware:

- The merging of the two most demanding AES operations into a single memory module, and the usage of the same hardware structure for encryption and decryption is proposed. The resulting AES core is capable of ciphering throughputs from 2 to 34Gbit/s, improving related art by more than 500% [70].
- A DES core capable of achieving ciphering throughputs of 1Gbit/s. Results also suggest an improvement of the Throughput per Slice metric by 20% with the use of memory blocks [79].
- To improve the SHA implementations, hardware rescheduling and reutilization techniques are proposed. Implementations results suggest improvements up to 100% regarding related art. Throughputs above 1Gbit/s are achieved [63, 64, 80].
- For the Whirlpool, the merging of the key computation and data compression, and the use of memory modules to compute part of the algorithm, suggest improvements of 160% regarding related art. Throughputs above 5Gbit/s are achieved.
- Improvement of modular multiplication units, used in asymmetrical encryption algorithms, through the employment of Residue Number Systems (RNS) is proposed, with the use of adaptable and more balanced moduli sets and the utilization of embedded binary multipliers. FPGA implementations suggest improvements of 30% regarding binary multiplications [81–84].

7.3 Proposed Research Directions

The following proposes some future research directions and improvements to the SCM.

- Given that the hardware units created to be reconfigured on the device are statically allocated on the device, during the *place&route* process, an interesting research direction is to have onboard hardware changing the allocation information in the bitstream. This might be accomplished by creating the bitstreams for a CrCU allocated in the first slot; during configuration time the offset is added to the frame address in the bit stream.

This will allow a dynamic allocation of the CrCU within the SCM. This offset can be added to the bitstream data by detecting when an instruction in the bitstream updates the Frame Address Register. When such instruction is detected the original frame address is added with the adequate value for the configuration to be shifted to the desired slot. This offset addition is performed after the CrCU bitstream is read into the hash function hardware and before the checking the region delimitation constrains of the attestation module. By calculating the hash value before the offset addition, the generated Digest Message is independent of the location were the CrCU is to be allocated. However, in order to assure that the CrCU is not written outside the reserved CrCU space, the actual address (with the offset) is used in the region delimitation hardware.

- To implement the Secure Computing Module on an FPGA with EPROM or any other type of non-volatile memory. This allows the SCM to be safely initialized and to securely store critical data in environments susceptible to physical tampering.

- To research the use of the proposed Secure Computing structure to implement the more user restricted Trusted Computing approach. In this case the user has a much more restricted use of the chip, namely, he is neither allowed to change the Endorsement Key nor has he the option to decide which features of the TC to use. To enforce the non alteration of the Endorsement Key, the FPGA must have internal non-volatile memory. This memory is then used to load the initialization configuration and critical data, like the Endorsement Key. The user will have no access or knowledge regarding the unchangeable Endorsement Key. The implementation of TC on reconfigurable devices requires the mandatory use of devices with internal non-volatile memory in order for the initial configuration to be self-loadable, and not to be altered.

- To develop software support for the Secure Computing Module in order to manage the keys and critical data in an efficient manner, and to mimic the behavior of a Trusted Platform Module (TPM), or to properly interact with higher level applications.

- Design a multi-region delimitation hardware or to have it mark which slots have been written on. This will avoid the need to consider all the reconfigurable area as attacked, in case a tampered bitstream is detected.

If an invalid bitstream is loaded, only the slots that have been written on, have to be considered invalid, not the whole region.

Appendix A

Improving RNS multiplication

Contents

In the last years, the Residue Number System (RNS) has become an important research field in the area of efficient digital computing [85]. A great amount of research has already been done to exploit the RNS properties for performing non-weighted, carry-free arithmetic. This characteristic can be greatly advantageous when repetitive arithmetic operations have to be performed, especially when large operators are involved. A known example of such operations is the multiply-accumulate operation for linear filtering and in Number Theoretic Transforms, typically found in Digital Signal Processing (DSP) [86, 87]. Another growing field where efficient multipliers for large operands are needed is in cryptography, namely in what concerns the asymmetrical encryption algorithm [88, 89], that intensively uses modular multiplications with operands up to 2048 bits, as described in Chapter 2.2; or the IDEA [90] symmetrical algorithm, which applies the $2^{16} + 1$ modular multiplication.

By splitting the computation of a large operand, into several smaller, non dependant channels, the performance can be improved. This approach also reduces the required area, especially in the multiplication units where the area grows with the square of the operands length [91]. Nevertheless, in RNS it has to be taken into account the need to perform forward and inverse conversions, which represent an unavoidable overhead, since the majority of the standards use the binary representation.

With the increase of the modulus size, combinatorial RNS architectures become more efficient than look-up table architectures [92]. Therefore, modulus using a nearby value to a power of two is preferable, since for the same technology only modified functional blocks of binary architectures have to be applied. This is the case not only of the commonly adopted 3-moduli set $\{2^n - 1, 2^n, 2^n + 1\}$, but also of the 4-moduli supersets that introduce a fourth element of the type $\{2^{n+1} \pm 1\}$ to increase the dynamic range [93]. Mathematical properties of these moduli sets are also important to develop efficient converters between the RNS and the binary number system, either using the Chinese Remainder Theorem (CRT) or the mixed-radix conversion technique [94].

This appendix is mainly focused in multiply operations in RNS; however, the methods and techniques herein proposed benefit the RNS arithmetic in general. From a detailed analysis of the RNS characteristics, the two main critical aspects that prevent form obtaining a higher speed-up are identified as:

- the differences in the complexity of the arithmetic units for the various modulus of the set are not negligible, in particular the longer critical path of the $2^n + 1$ moduli set;

- in spite of just exceeding by one value the power of two, the additional bit of the moduli $2^n + 1$ multiplication, originates a more complex computational structure and consequently the processing time increase, when compared with the other moduli multiplications.

These two aspects stress the unbalancing between the elements, of not only the traditional moduli set but also, the extended moduli sets and degrade the performance of the RNS. In this thesis, two independent but complementary techniques are proposed to tackle this problem: the enhancement of the arithmetic unit for the modulo $2^n + 1$ multiplication [81] and a modification of the moduli sets to obtain more balanced structures. This last proposed modification corresponds to the overloading of the binary base element (channel), which allows the alleviation of the other non binary channels for a given dynamic range.

In order to evaluate the improvements obtained from the proposed modulo $2^n + 1$ multiplication structure and the overloading of the binary channel, the arithmetic units for the modified and more balanced moduli set were implemented. Most of the results presented in this appendix were obtained for ASIC technology, given that most of the related art is also presented for this type of technology; however, results for FPGA technology were also obtained, since this is the target technology of this thesis.

The ASIC implementation was performed on a high density StdCell library from *UMC*, based on a 0.13μm CMOS process from *Virtual Silicon Technology Inc.* [95], using the *Synopsys* synthesis tools. The FPGA technology used was the Xilinx Virtex II Pro.

Experimental results suggest the advantages of applying the proposed $2^n + 1$ multiplier and the balanced moduli set. Improvements for the multiplication units and the RNS arithmetic in general up to 32%, for the ASIC implementations. In FPGA, results suggest improvements of 20%.

In the following two sections, the enhanced 2^n+1 multipliers and the extension of the binary channel with new converters are proposed. The experimental results section concludes the RNS work presented in this thesis.

A.1 Enhanced modulo $2^n + 1$ multipliers

Unlike multiplication modulo $2^n - 1$ or modulo 2^n, the multiplication modulo $2^n + 1$ has to accommodate the cases where one or both operands are equal to 2^n. This is the main reason for the modulo 2^n+1 multipliers being the slowest ones in both the 3 and the 4-moduli sets. The following proposes improved modulo $2^n + 1$ multiplication structures for fine grained technologies, like ASIC; and for coarse grained technologies, like FPGAs with embedded binary multipliers.

A.1.1 Fine grained modulo $2^n + 1$ multipliers

As recently shown, although the modified Booth recoding can be applied for designing modulo 2^n+1 multipliers, it usually does not leads to faster multipliers nor significantly reduces the required hardware [82]. This thesis proposes to enhance one of the most efficient modulo 2^n+1 multiplication architectures proposed by Zimmermann [96], which does not use Booth recoding. However, by manipulating the values for the particular cases when none or both operands are equal to zero, the proposed structure seeks to perform this calculation in a more efficient way, by speeding-up the computation without significantly increasing the required circuit area.

Let us consider an integer number X represented with $n + 1$ bits:

$$X = \sum_{i=0}^{n} 2^i x_i = 2^n x_n + \sum_{i=0}^{n-1} 2^i x_i = 2^n x_n + X' \tag{A.1}$$

and the multiplication of numbers X and Y, which can be computed as:

$$\begin{aligned} \langle X \times Y \rangle_{2^n+1} &= \big\langle 2^n x_n \cdot 2^n y_n + 2^n x_n \cdot Y' + \\ &+ 2^n y_n \cdot X' + X' \cdot Y' \big\rangle_{2^n+1} . \end{aligned} \tag{A.2}$$

By exploiting the following properties:

$$\langle 2^n \rangle_{2^n+1} = \langle -1 \rangle_{2^n+1} \Rightarrow \langle 2^n X \rangle_{2^n+1} = \big\langle \overline{X} + 2 \big\rangle_{2^n+1} \tag{A.3}$$

(A.2) can be rewritten as:

$$\langle X \times Y \rangle_{2^n+1} = \left\langle \underbrace{x_n \cdot y_n}_{P_3} \underbrace{- x_n \cdot Y' - y_n \cdot X'}_{P_2} \underbrace{+ X' \cdot Y'}_{P_1} \right\rangle_{2^n+1} . \tag{A.4}$$

In the modulo $2^n + 1$ representation, when $a_n = 1$ the other bits are zero ($A' = 0$) and consequently the multiplication can be divided in three distinct operations:

i) for $x_n = 0$ and $y_n = 0$:

$$\langle X \times Y\rangle_{2^n+1} = \langle P_1\rangle_{2^n+1} = \left\langle X' \cdot Y'\right\rangle_{2^n+1} \; ; \tag{A.5}$$

ii) when one and only one of the nth bits is equal to '1':

$$\begin{aligned}\langle X \times Y\rangle_{2^n+1} = \langle P_2\rangle_{2^n+1} &= \left\langle 2^n y_n \cdot X' + 2^n x_n \cdot Y'\right\rangle_{2^n+1} \\ &= \left\langle y_n \cdot (-X') + x_n \cdot (-Y')\right\rangle_{2^n+1} \\ &= \left\langle y_n \cdot \overline{X'} + 2 + x_n \cdot \overline{Y'} + 2\right\rangle_{2^n+1} \\ &= \left\langle y_n \cdot \overline{X'} + x_n \cdot \overline{Y'} + 4\right\rangle_{2^n+1} \; ;\end{aligned} \tag{A.6}$$

iii) and finally when the two nth bits of X and Y are '1':

$$\langle X \times Y\rangle_{2^n+1} = \langle P_3\rangle_{2^n+1} = \langle 2^n \cdot 2^n\rangle_{2^n+1} = \langle 1\rangle_{2^n+1} \, . \tag{A.7}$$

Computation of $\langle P_1\rangle_{2^n+1}$:

To compute (A.5), it can be applied the approach presented in [96]:

$$\begin{aligned}\langle P_1\rangle_{2^n+1} &= \left\langle X' \cdot Y'\right\rangle_{2^n+1} \\ &= \left\langle \sum_{i=0}^{n-1} 2^i x_i \cdot Y' \right\rangle_{2^n+1} \\ &= \left\langle \sum_{i=0}^{n-1} x_i \cdot \left\langle 2^i Y'\right\rangle_{2^n+1} \right\rangle_{2^n+1} \\ &= \left\langle \sum_{i=0}^{n-1} (x_i \cdot y_{n-i-1} \cdots y_0 \, \overline{y}_{n-1} \cdots \overline{y}_{n-i} + 1) + 2 \right\rangle_{2^n+1} \\ &= \left\langle \sum_{i=0}^{n-1} (PP_i + 1) + 2 \right\rangle_{2^n+1}\end{aligned} \tag{A.8}$$

Where in (A.8) it can be considered that modulo $2^n + 1$ adders intrinsically add an extra unit. Instead of using a multiplication matrix with $n \times n$ bits inputs for adding the partial products (PP_i), followed by a Carry Save Adder

(CSA) to add the constant '2' in (A.8) [96], the multiplication matrix can be redesigned to encompass an extra input where such constant is directly introduced. This is advantageous when multipliers are implemented using Wallace tree adders, since for most values (n) the delay is approximately the same as for $n+1$ inputs: the number of Full Adders (FA) in the critical path of a n-bit Wallace tree is given by $W(n{+}1) = \lfloor \frac{3}{2} W(n) \rfloor$ [97], as presented in Table A.1. However, this is only a rough approximation of the computation delay not taking into account the wire propagation delay, which is becoming increasingly more significant in nowadays technologies.

Table A.1: Number of FA stages in a Wallace-tree structure.

j	3	4	5-6	7-9	10-13	14-19	20-28	29-42	43-63	...
WT(j)	1	2	3	4	5	6	7	8	9	...

Computation of $\langle P_3 \rangle_{2^n+1}$:

The computation of (A.7) is rather simple by taking into consideration that when $x_n = y_n = 1$ the partial results P_1 and P_2 are null, and thus the final product is '1'. This particular case can be accommodated simply by setting the least significant bit of the multiplication to '1' whenever $x_n = y_n = 1$ (this operation can be efficiently performed using a simple OR gate). Since most efficient modulo adders have the result of the LSB calculated before the MSB [96,98], the OR gate should not have any implication in the computation time.

Computation of $\langle P_2 \rangle_{2^n+1}$:

The most significant improvement proposed for the multiplication unit concerns the introduction of the partial product described in (A.6) into the final result. In [96], Zimmermann adopted a multiplexer to select the result based on the fact that when the result of (A.6) is not null (A.5) is zero, and vice-versa. Such multiplexer is located at the output of the parallel multiplier, thus contributing to the critical path of the circuit. Our proposal is to introduce this partial product directly in the multiplication matrix, by adding extra inputs in the Wallace tree. However, each extra input line added to the matrix of adders, corresponds to introduction of an extra modulo $2^n + 1$ CSA, which intrinsically adds an extra unit [98]. Consequently, the constant initially introduced in the multiplication matrix has to be corrected in order to take into consideration both the extra inputs and the constant required by the partial product (P_2) in (A.6). The two elements of P_2 are introduced in distinct inputs of the

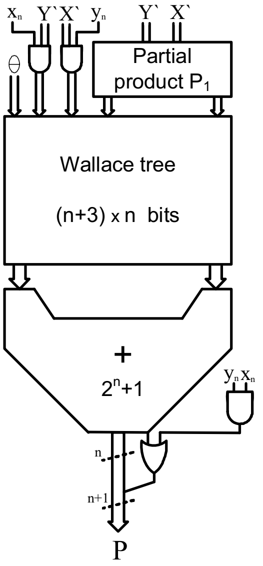


Figure A.1: Modulo 2^n+1 multiplier

adders' matrix, which requires the addition of 2 extra units and results in:

$$\langle P_1+P_2\rangle_{2^n+1} = \left\langle \sum_{i=0}^{n-1}(PP_i+1) + 2 + \underbrace{y_n \cdot \overline{X}'}_{PP_n} + \underbrace{x_n \cdot \overline{Y}'}_{PP_{n+1}} + 4 \right\rangle_{2^n+1} =$$

$$\langle P_1+P_2\rangle_{2^n+1} = \left\langle \sum_{i=0}^{n-1}(PP_i+1)+(PP_n+1)+(PP_{n+1}+1)+4 \right\rangle_{2^n+1} =$$

$$\langle P_1+P_2\rangle_{2^n+1} = \left\langle \sum_{i=0}^{n+1}(PP_i+1)+4 \right\rangle_{2^n+1}, \tag{A.9}$$

where the first constant term '2' of the equation is intrinsically added. The resulting architecture is depicted in Figure A.1.1, where θ represents the added constant.

A.1.2 Coarse grained modulo $2^n + 1$ multipliers

In coarse grained technologies, like FPGA, where embedded multipliers are available, the implementation of fine grained multiplication structures is less efficient. The embedded multipliers are already optimized, yielding in better performances. As such, it is preferable to implement the multiplication structures using these embedded structures. However, only binary multipliers are typically available. The following presents modulo $2^n + 1$ multiplication units based on the proposed computation structure of [99]. A computational approach to the implementation of multipliers, when the operands width does not allow the direct use of the embedded binary multipliers, is also proposed.

In the strait forward implementation of modulo 2^n+1 multipliers, the operands are first multiplied in binary, followed by a modulo $2^n + 1$ reduction. This calculation is described as:

$$\begin{aligned}\langle X \times Y\rangle_{2^n+1} &= \langle M\rangle_{2^n+1}\\ &= \left\langle 2^{2n}M_2 + 2^n M_1 + M_0\right\rangle_{2^n+1}\\ &= \langle M_2 - M_1 + M_0\rangle_{2^n+1}\\ &= \left\langle M_2 + \overline{M_1} + 2 + M_0\right\rangle_{2^n+1}. \end{aligned} \tag{A.10}$$

Since, M_2 is only equal to '1' when $X = Y = 2^n$, implying that the parcel M_0 and M_1 are equal to zero, the value M_2 can be added in the final result by a OR gate, as described for the previous multiplier. Given that each modulo $2^n + 1$ adder, adds one unit to the final result, the multiplication can be performed as:

$$\langle X \times Y\rangle_{2^n+1} = \left\langle \langle \overline{M_1} + M_0 + 1\rangle_{2^n+1} + 1\right\rangle_{2^n+1} \ or\ M_2. \tag{A.11}$$

This type of structure can only be used when binary multipliers have outputs as big as the operand M. Binary multipliers with the size of the operand M can be realized from several smaller binary multipliers. However, in this case, a more efficient multiplication modulo 2^n+1 structure can be developed, from partial binary multiplications.

The following describes the implementation of an improved modulo $2^n + 1$ multiplier, for n=32, using binary multiplication units with 16 and 17 bit inputs, and 32-bit outputs.

As stated above, when the nth bit of the operand is '1' the remaining bits are '0'. Thus, when multiplying the higher 17 bits of each operand the result can be expressed in a 32 bit value, except for the case when both input operand are equal to 2^n. In this case the result is equal to 2^{2n} as described in (A.7). With this, it can also be concluded that the multiplication of a 16-bit operand by a 17-bit operand is smaller than 2^{2n} thus, it can be expressed by a 32-bit value.

With $m = \frac{n}{2} = 16$, the multiplication modulo $2^n + 1$ can be described as:

$$\begin{aligned}
\langle X \times Y \rangle_{2^n+1} &= \langle (2^m X_{32-16} + X_{15-0}) \times Y \rangle_{2^n+1} \\[1em]
&= \langle 2^n (X_{32-16} \times Y_{32-16}) + X_{15-0} Y_{15-0} + \\
&\quad +2^m (X_{32-16} \times Y_{15-0} + X_{15-0} \times Y_{32-16}) \rangle_{2^n+1} \\[1em]
&= \big\langle 2^n X_H Y_H + 2^{2n} x_n y_n + X_L Y_L + \\
&\quad +2^m X_H Y_L + 2^m X_H Y_L \rangle_{2^n+1} \\[1em]
&= \langle \quad -X_H Y_H + x_n y_n + X_L Y_L + \\
&\quad \left. +2^m \underbrace{X_H Y_L}_{A} + 2^m \underbrace{X_L Y_H}_{B} \right\rangle_{2^n+1} \\[1em]
&= \big\langle \overline{X_H Y_H} + 2 + x_n y_n + X_L Y_L + 2^m A + 2^m B \big\rangle_{2^n+1}
\end{aligned} \tag{A.12}$$

The value A of (A.12) can be computed by (A.13).

$$\begin{aligned}
\langle 2^m X_H Y_L \rangle_{2^n+1} &= \langle 2^m A \rangle_{2^n+1} \\
&= \langle 2^m (2^m A_H + A_L) \rangle_{2^n+1} \\
&= \langle 2^n A_H + 2^m A_L \rangle_{2^n+1} \\
&= \langle -A_H + 2^m A_L \rangle_{2^n+1} \\
&= \Big\langle \overline{(0 \cdots 0 a_{31} \cdots a_{16})} + 2 + (a_{15} \cdots a_0 0 \cdots 0) \Big\rangle_{2^n+1} \\
&= \left\langle (1 \cdots 1 0 \cdots 0) + 2 + \underbrace{(a_{15} \cdots a_0 \overline{a_{31} \cdots a_{16}})}_{A'} \right\rangle_{2^n+1} \\
&= \big\langle 2^n - 2^m + 2 + A') \big\rangle_{2^n+1}
\end{aligned} \tag{A.13}$$

Applying (A.13) to the parcel B, (A.12) can be rewritten as:

$$\begin{aligned}
&\big\langle \overline{X_H Y_H} + 2 + x_n y_n + X_L Y_L + A' + B' + 2 \times (2^n - 2^m + 2) \big\rangle_{2^n+1} \\
&= \big\langle \overline{X_H Y_H} + 2 + x_n y_n + X_L Y_L + A' + B' + 2^n - 2^{m+1} + 3) \big\rangle_{2^n+1} \\
&= \big\langle \overline{X_H Y_H} + x_n y_n + X_L Y_L + A' + B' + (2^n - 2^{m+1} + 1) + 4 \big\rangle_{2^n+1}
\end{aligned} \tag{A.14}$$

As stated before, each modulo $2^n + 1$ adder adds one extra unit to the final result, thus (A.14) can be written as:

$$\begin{aligned} \Big\langle \Big\langle \Big\langle & \langle (\overline{X_H Y_H} + X_L Y_L + A') + 1 \rangle_{2^n+1} \\ & +(B') + 1 \Big\rangle_{2^n+1} \\ & +(2^n - 2^{m+1} + 1) + 1 \Big\rangle_{2^n+1} + x_n y_n + 1 \Big\rangle_{2^n+1} \end{aligned} \tag{A.15}$$

In this structure CSA modulo $2^n + 1$ are used for the intermediate addition and a full modulo $2^n + 1$ for the final addition. It should also be noted that, the addition of the constant can be hardwired, resulting in a simplified CSA adder. Also the value $x_n y_n$ can be added to the result by an OR gate at the end of the result to the bit '0'.

This computation can be performed by 3 modulo $2^n + 1$ CSA for the intermediate additions, followed by a full modulo $2^n + 1$ adder, resulting in a more compact $2^n + 1$ multiplication structure, as depicted in Figure A.1.2.

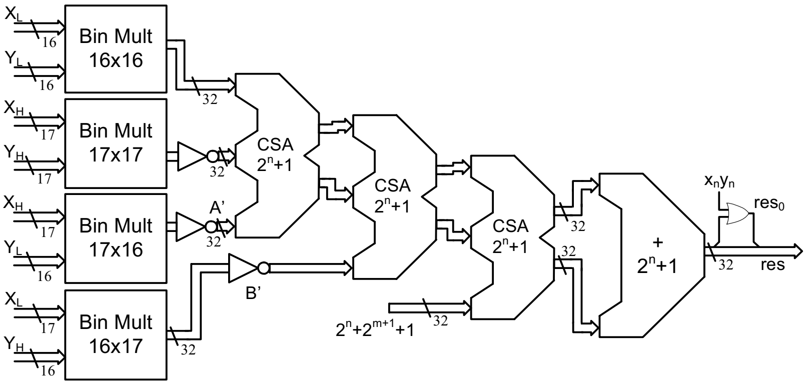


Figure A.2: Optimized FPGA modulo $2^n + 1$ multiplier

A.2 Balanced RNS moduli sets

As stated before, the non-binary channels exhibit a lower performance than the corresponding binary ones, especially the modulo $2^n + 1$. This has a negative impact in the efficiency of RNS, as shown in the result section. In this section, the overloading of the binary channel is proposed to balance the computation time for the various channels, especially the $2^n + 1$ multiplication which is where the critical path is commonly located. The encoder and the decoder are the only new arithmetic units required by these new moduli sets, since the addition and the multiplication units for the traditional moduli set can be directly used.

This section is organized as follows: *i*) the correctness of the proposed binary channel overloading technique is proved and *ii*) the encoder and the decoder units for the proposed modified 3 and 4-moduli sets are presented.

A.2.1 Binary channel extension

Recently, moduli sets with larger binary channels have been proposed, mainly for increasing the dynamic range [100]. The width of the binary channel was doubled in order to increase the dynamic range to approximately the value achieved with 4-moduli supersets. In this thesis, the width of the binary channel is increased by a variable factor 2^k, in order to overcome the relative difference in the complexity of the arithmetic units. The new class of moduli sets is comprehensive, in the sense that it also integrates the traditional 3-moduli sets and 4-moduli supersets. Although k can take any general value $k > -n$, we only consider values with practical interest: $k \geq 0$. Therefore, the following general new class of moduli sets is defined:

$$\begin{gathered}\{2^n - 1, 2^{n+k}, 2^n + 1[, 2^{n+\gamma} + \alpha]\} \\ for \quad k \in \mathbb{N}_0 \ ; \ \gamma, \alpha = \pm 1\end{gathered} \tag{A.16}$$

which includes the traditional 3-moduli and 4-moduli sets, for $k = 0$. It is worth noticing that, the binary channel is the one that requires the simplest arithmetic units. For example, for fast adders based on binary tree structures the increase of, at most, n bits leads to the introduction of just one further level in the tree, which corresponds to the additional stage usually required by the corresponding modulo $2^n + 1$ adders. In this proposal, the value k is variable and can be adjusted according to the technology and the required dynamic range.

Let us prove the validity of the new class of moduli sets, which corresponds to prove that all the elements of the moduli set are relative primes.

Validity proof of the Moduli Sets

For proving that $2^n - 1$, $2^n + 1$, $2^{n+\gamma} + \alpha$ $(\gamma, \alpha = \pm 1)$ and 2^{n+k} $(k \in \mathbb{N})$ are relative primes it is only necessary to prove that 2^{n+k} is a relative prime to each of the other elements, since it has already been proved that all the other elements are relative primes, e.g. see [93, 101].

Lemma 1 : $(2^{n+k}, 2^n - 1)$, $(2^{n+k}, 2^n + 1)$, $(2^{n+k}, 2^{n+1} + 1)$, $(2^{n+k}, 2^{n+1} - 1)$, $(2^{n+k}, 2^{n-1} + 1)$ and $(2^{n+k}, 2^{n-1} - 1)$ are pairwise relative prime numbers for $k \in \mathbb{N}$.

Proof: The Euclidian algorithm for computing the greatest common divisor of two numbers $(\gcd(a, b))$ can be used to prove that a pair of numbers (a,b) are relative primes.

$$\begin{aligned} &r_1 =< a >_b; r_2 =< b >_{r_1}; \ldots; r_{k+1} =< r_{k-1} >_{r_k} \\ &r_{k+1} =< r_{k-1} >_{r_k} = 0 \Rightarrow \mathbf{gcd(a, b) = r_k} \end{aligned} \tag{A.17}$$

The six pairs of numbers in lemma A.2.1 can be reduced to only two different types, simply by replacing the variables. Therefore, to prove lemma A.2.1 one needs only to compute the $\gcd$ for the following two pairs of numbers: $(2^{n+k}, 2^n - 1)$ and $(2^{n+k}, 2^n + 1)$, $k \in \mathbb{N}_0$.

For $2^n - 1 \quad (\varphi < n; \phi \in \mathbb{N})$:

$$\begin{aligned} < 2^{n+k} >_{2^n-1} &=< 2^{\phi n + \varphi} >_{2^n-1} \\ &= 2^{\varphi}; \\ < 2^n - 1 >_{2^{\varphi}} &=< -1 >_{2^{\varphi}} \\ &= 2^{\varphi} - 1; \\ < 2^{\varphi} >_{2^{\varphi}-1} &= 1; \\ \gcd(2^{n+k}, 2^n - 1) &= 1 \quad , \end{aligned} \tag{A.18}$$

and for $2^n + 1 \quad (\varphi < n; \phi \in \mathbb{N})$:

$$\begin{aligned} < 2^{n+k} >_{2^n+1} &=< 2^{\phi n + \varphi} >_{2^n+1} \\ &=< (2^n)^{\phi} \times 2^{\varphi} >_{2^n+1} \\ &=< (-1)^{\phi} \times 2^{\varphi} >_{2^n+1}; \end{aligned} \tag{A.19}$$

for ϕ odd:

$$\begin{aligned} < -2^{\varphi} >_{2^n+1} &= 2^n + 1 - 2^{\varphi}; \\ < 2^n + 1 >_{2^n+1-2^{\varphi}} &= 2^{\varphi}; \\ < 2^n + 1 - 2^{\varphi} >_{2^{\varphi}} &= 1; \end{aligned} \tag{A.20}$$

for ϕ even:

$$\begin{aligned} &< 2^{\varphi} >_{2^n+1} = 2^{\varphi}; \\ &< 2^n + 1 >_{2^{\varphi}} = 1; \end{aligned} \tag{A.21}$$

thus:

$$\gcd(2^{n+k}, 2^n + 1) = 1 \quad . \tag{A.22}$$

From (A.18) and (A.22) it can be concluded that $(2^{n+k}, 2^n - 1)$, $(2^{n+k}, 2^n + 1)$, $(2^{n+k}, 2^{n+1} - 1)$, $(2^{n+k}, 2^{n+1} + 1)$, $(2^{n+k}, 2^{n-1} - 1)$ and $(2^{n+k}, 2^{n-1} + 1)$ are pairwise of relative prime numbers for $n, k \in \mathbb{N}$. Q.E.D.

A.2.2 New conversion units

In order to implement the binary channel overloading technique, encoder and decoder memoryless units for the new sub-class of 3-moduli sets $\{2^n - 1, 2^{n+k}, 2^n + 1\}$ are proposed. Values of k such as $0 < k \leq n$, where $n \in \mathbb{N}$, are considered, as well as the standard and the diminished-1 number representations.

Binary to RNS Converters

An integer X in the range $[0, M - 1]$, represented as:

$$X = \sum_{i=0}^{4n-1} X_i 2^i = N_3 2^{3n} + N_2 2^{2n} + N_1 2^n + N_0, \tag{A.23}$$

where N_3 is a variable length value, with k bits, represented as:

$$N_3 = \{X_{3n+k-1} \cdots X_{3n}\} \qquad \text{for } 1 \leq k \leq n \tag{A.24}$$

can be uniquely represented in RNS by the 3-tuple $\{x_1, x_2, x_3\}$ for the moduli set $\{2^n - 1, 2^{n+k}, 2^n + 1\}$ ($\{m_1, m_2, m_3\}$).

The dynamic range of this moduli takes the value M given by (A.25).

$$M = \prod_{i=1}^{3} m_i = (2^n + 1) \times (2^{n+k}) \times (2^n - 1) = 2^{3n+k} - 2^{n+k}. \tag{A.25}$$

In order to obtain the RNS representation of the integer X, three independent converters, one for each channel, are required. The simplest converter is the one for the m_2 channel, because for the $2^n - 1$ and $2^n + 1$ channels the calculation of the corresponding residues depends on the value of all the bits of X.

Converter for the $\mathbf{2^{n+k}}$ Channel

The value x_2 can be obtained by the remainder of the division of X by 2^{n+k}, which can be accomplished by truncating the $(n + k)$ less significant bits of X:

$$x_2 = \langle X \rangle_{2^{n+k}} = X_{n+k-1} \cdots X_0 \ . \tag{A.26}$$

Converter for the $\mathbf{2^n - 1}$ Channel

Instead of using a division operation to calculate the $2^n - 1$ residue, which is a complex and expensive operation, both in terms of area and speed, this calculation can be performed as a sequence of additions:

$$\begin{aligned} x_1 &= \langle X \rangle_{2^n-1} \qquad \text{(A.27)} \\ &= \left\langle N_3 2^{3n} + N_2 2^{2n} + N_1 2^n + N_0 \right\rangle_{2^n-1} \ . \end{aligned}$$

By taking into account that

$$< 2^{k \times n} >_{2^n-1} = < 1 >_{2^n-1} \ , \tag{A.28}$$

(A.27) can be rewritten as:

$$x_1 \ = \ < N_3 + N_2 + N_1 + N_0 >_{m_1} \ . \tag{A.29}$$

Hence, the conversion to the $2^n - 1$ channel can be performed simply by adding modulo $2^n - 1$ the several segments of X. However, in order to perform these four additions one does not need to use four modulo $2^n - 1$ carry propagate adders, which would be slow and inefficient. As represented in Figure A.3, it is possible to group the values to be added as:

$$x_1 = << N_3 + < N_2 + N_1 + N_0 >_{m_1} >_{m_1} >_{m_1} \tag{A.30}$$

with the first and second additions ($< N_2 + N_1 + N_0 >_{2^n-1}$ and $< N_3 + S_1 + C_1 >_{2^n-1}$, respectively), being performed by a 3-2 compressor (CSA), without a significant increase neither in the delay nor in the area. Nevertheless, the third and last modulo $2^n - 1$ addition ($< S_2 + C_2 >_{2^n-1}$), requires a modulo $(2^n - 1)$ FA.

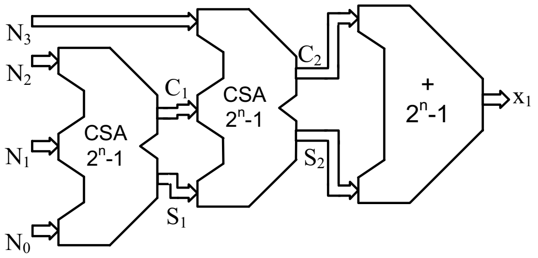


Figure A.3: Modulo (2^n-1) Binary to RNS converter

Converter for the 2^n+1 Channel

Similarly, the 2^n+1 residue can be calculated as:

$$\begin{aligned} x_3 &= \langle X \rangle_{2^n+1} \\ &= \left\langle N_3 2^{3n} + N_2 2^{2n} + N_1 2^n + N_0 \right\rangle_{2^n+1} . \end{aligned} \tag{A.31}$$

Since:

$$< 2^n >_{2^n+1} = < -1 >_{2^n+1}, \tag{A.32}$$

(A.31) can be simplified to:

$$x_3 = < -N_3 + N_2 - N_1 + N_0 >_{2^n+1} . \tag{A.33}$$

The computation of (A.33) can be further simplified by using only additions, given that a modulo 2^n+1 subtraction can be expressed as:

$$\begin{aligned} < -N_i >_{2^n+1} &= < (2^n+1) - N_i >_{2^n+1} \\ &= < \overline{N_i} + 2 >_{2^n+1} \end{aligned} \tag{A.34}$$

and therefore (A.33) can be rewritten as:

$$x_3 = < \overline{N_3} + 2 + N_2 + \overline{N_1} + 2 + N_0 >_{2^n+1} . \tag{A.35}$$

In the standard representation, the value x_3 represents the 2^n+1 residue of value X, instead of value $X-1$. As shown in the following, the conversion unit for the standard representation has a higher complexity.

Like in the 2^n-1 modulo conversion, the additions can be grouped and partially added by 3-2 compressors. Nevertheless, the constant value (4) has to be

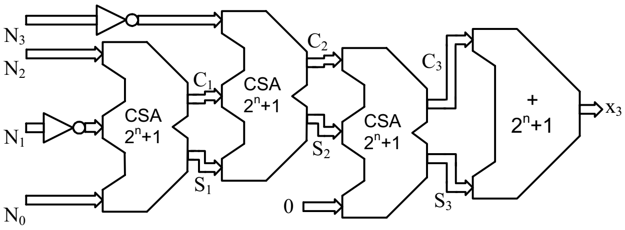


Figure A.4: Modulo (2^n+1) Binary to RNS converter

added to compute (A.35). However, as referred before, modulo 2^n+1 compressors not only adds the input values, but also intrinsically adds one extra unit [98]. Therefore, by using this characteristic of the 2^n+1 modulo adders, the value x_3 can be computed as:

$$\begin{aligned} x_3 &= <1+<1+<1+\overline{N_3}+<1+N_2+ \\ &\quad \overline{N_1}+N_0>_{2^n+1}>_{2^n+1}>_{2^n+1}>_{2^n+1} . \end{aligned} \tag{A.36}$$

The binary to modulo 2^n+1 converter is depicted in Figure A.4, where it can be noticed that the last CSA is only used to add one unit to S_2+C_2+0, resulting in a simplified 2-2 compressor.

In the diminished-1 representation, the remainder of the division is represented as $x_3 =< X-1 >_{2^n+1}$, where the obtained value is always the correct value minus one. The computation of x_3 for this representation, using a similar approach to the one used in (A.35), results in the following expression:

$$\begin{aligned} x_3 &= < X-1 >_{2^n+1} \\ &= < \overline{N_3}+2+N_2+\overline{N_1}+1+N_0 >_{2^n+1} . \end{aligned} \tag{A.37}$$

Decomposing the addition, (A.37) can be rewritten as:

$$\begin{aligned} x_3 &= <1+<1+\overline{N_3}+<1+N_2 \\ &\quad +\overline{N_1}+N_0>_{2^n+1}>_{2^n+1}>_{2^n+1}, \end{aligned} \tag{A.38}$$

which corresponds to the hardware structure presented in Figure A.5, that is even simpler than the one depicted in Figure A.4 for the standard representation.

RNS to Binary Converters

The proposed RNS to Binary memoryless converter is based on the Chinese Remainder Theorem (CRT) [83, 98, 101]:

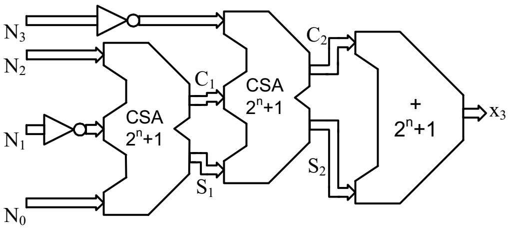


Figure A.5: Modulo (2^n+1) Binary to diminished-1 RNS converter

$$X = \left\langle \sum_{i=1}^{3} \hat{m}_i \left\langle \frac{x_i}{\hat{m}_i} \right\rangle m_i \right\rangle_M , \tag{A.39}$$

where $\hat{m}_i = M/m_i$ and $\left\langle \frac{1}{\hat{m}_i} \right\rangle_{m_i}$ is the multiplicative inverse of $\hat{m}_i$ and M is given by (A.25). Equation (A.39) can be written as [98]:

$$X \; + MZ(X) = \sum_{i=1}^{3} \hat{m}_i \left\langle \frac{1}{\hat{m}_i} \right\rangle_{m_i} x_i, \tag{A.40}$$

where $Z(x)$ is a non negative integer, function of X. For the proposed moduli set $\{m_1 = 2^n - 1, m_2 = 2^{n+k}, m_3 = 2^n + 1\}$, the multiplicative inverse of $\hat{m}_i$ $(i = 1, 2, 3)$ is [83]:

$$\left\langle \frac{1}{\hat{m}_1} \right\rangle_{m_1} = 2^{n-1} \tag{A.41a}$$

$$\left\langle \frac{1}{\hat{m}_2} \right\rangle_{m_2} = 2^{n+k} - 1 \tag{A.41b}$$

$$\left\langle \frac{1}{\hat{m}_3} \right\rangle_{m_3} = 2^{n-1}. \tag{A.41c}$$

By replacing these values in (A.40):

$$\begin{aligned} X \; + MZ(X) = \quad & 2^{n+k}(2^n - 1)(2^{n-1})x_3 \\ & + (2^{2n} - 1)(2^{n+k} - 1)x_2 \\ & + 2^{n+k}(2^n + 1)(2^{n-1})x_1 \end{aligned} \tag{A.42}$$

and dividing X by 2^{n+k}:

$$X = 2^{n+k} \left\lfloor \frac{X}{2^{n+k}} \right\rfloor + x_2, \tag{A.43}$$

where:

$$\left\lfloor \frac{X}{2^{n+k}} \right\rfloor = \Bigg\langle \underbrace{\left\langle (2^{2n-1} + 2^{n-1}) x_1 \right\rangle_{2^{2n}-1}}_{F} - 2^n x_3 + \underbrace{\left\langle (2^{2n} - 2) x_2 \right\rangle_{2^{2n}-1}}_{G} + \underbrace{\left\langle (2^{2n-1} + 2^{n-1}) x_3 \right\rangle_{2^{2n}-1}}_{H} \Bigg\rangle_{2^{2n}-1} . \quad \text{(A.44)}$$

Finally, with the simplification of the partial expressions F, G, H, $-2^n x_3$:

$$\begin{aligned} F &= x_{1,0} x_{1,n-1} \dots x_{1,1} x_{1,0} x_{1,n-1} \dots x_{1,1} \\ G &= \overline{x}_{2,n+k-1} \dots \overline{x}_{2,0} 1 \dots 1 \\ H &= x_{3,x} x_{3,n-1} \dots x_{3,1} x_{3,x} x_{3,n-1} \dots x_{3,1} \\ -2^n x_3 &= -r_3 = \overline{x}_{3,n-1} \dots \overline{x}_{3,0} 1 \dots 1 \overline{x}_{3,n}, \end{aligned} \quad \text{(A.45)}$$

the overlapping bits $x_{3,n}$ and $x_{3,0}$ are merged together and represented by $x_{3,x} = (x_{3,n} \ or \ x_{3,0})$. This is only possible due to the fact that $x_{3,n}$ and $x_{3,0}$ are mutually exclusive and cannot be simultaneously equal to '1'.

The upper 2^{n+k} bits of X can be calculated using two 3-2 compressors and one full adder modulo $2^{2n} - 1$:

$$\begin{aligned} \left\lfloor \frac{X}{2^{n+k}} \right\rfloor &= \langle H + G + F + (-2^n x_3) \rangle_{2^{2n}-1} \\ &= \left\langle \langle H + G + F \rangle_{2^{2n}-1} + (-2^n x_3) \right\rangle_{2^{2n}-1} . \end{aligned} \quad \text{(A.46)}$$

As it is observed in (A.46), the conversion from RNS to binary using the new extended moduli set can be performed by adding modulo $2^{2n} - 1$ the four partial expressions in (A.45), directly obtained from the 3 RNS channels. In order to optimize the converter, only one full adder (modulo $2^{2n} - 1$) is used in the final stage of the computation. The other additions are performed by two modulo $2^{2n} - 1$ CSAs, as depicted in Figure A.6, resulting in a RNS decoder with exactly the same structure as the RNS decoder for the traditional moduli set [101].

To decode a RNS value encoded in the diminished-1 representation, the value from the $2^n + 1$ channel (x_3) has to be incremented by one, which is equivalent to add $2^{2n-1} - 2^{n-1}$ modulo $2^{2n} - 1$ in (A.44). This extra value can be added

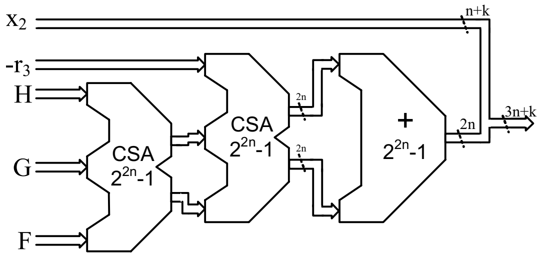


Figure A.6: RNS decoder for the standard representation

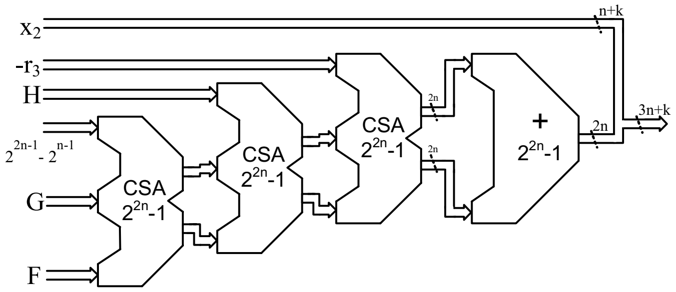


Figure A.7: RNS decoder for the diminished-1 representation

by an extra simplified CSA, in which one of the inputs is a constant value. As a result, only an extra HA is introduced in the critical path of the decoder, as depicted in Figure A.7.

Note that the complexity of the RNS to binary conversion for the new moduli sets is exactly the same as for the traditional moduli set. The only difference lays in the partial expression F, that in this new moduli set has no constant terms [101, 102].

A.3 Experimental results

This section presents the implementation results for the enhanced modulo $2^n + 1$ multipliers and the backward and forward converters proposed for the new moduli sets on the 0.13 μm standard cell technology, using the *Synopsys* synthesis tools; and the results on FPGA for the RNS multiplication. The obtained experimental results allow to individually evaluate the real impact of the proposed techniques, as well as the efficiency of the enhanced multipliers applied to the modified RNS moduli set.

A.3.1 ASIC enhanced modulo $2^n + 1$ multipliers

Figure A.8 depicts the ratio of the processing time between the proposed modulo $2^n + 1$ multiplier and the one proposed by Zimmermann in [96]. From this figure it can be observed that the proposed multiplication architecture significantly improves the processing time. However, there are exceptions, as is the case for 37 bits. In this particular situation, the Zimmermann architecture has a better performance, due to the advantages provided by the Wallace tree characteristics, not being so significant for such width. The new multiplication architecture is, on average, 9% better than the one proposed by Zimmermann [96]. In fact, for bigger word lengths the improvement obtained by this multiplication architecture is, on average, above 10% (see Table A.2).

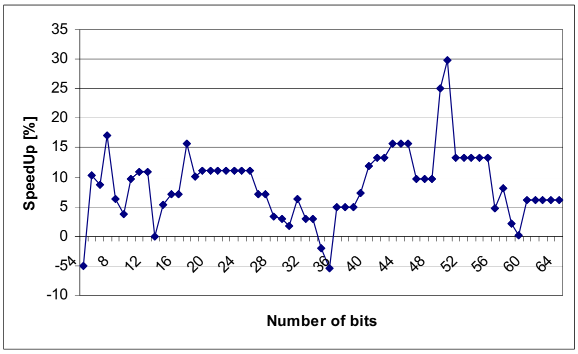


Figure A.8: Relative delay of the new multiplier regarding to the multiplier proposed by Zimmermann

Table A.2: Average improvement for the standard multiplication.

Number of bits	SpeedUp	Improvement [%]
4 - 64	1.1	9
40 - 64	1.12	11

The approach taken to introduce the partial product P_2 in the new multiplication architecture leads to an increase of the circuit area. This difference becomes less significant as the number of bits increases (see Figure A.9). This is only a significant disadvantage, in comparison to the architecture proposed by

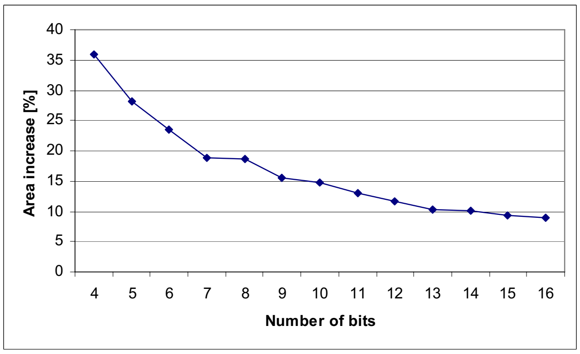


Figure A.9: Relative circuit area of the new multiplier: word length bellow 16 bits

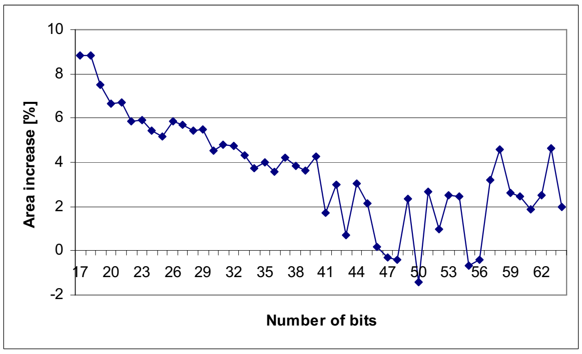


Figure A.10: Relative circuit area of the new multiplier: word length above 16 bits

Zimmermann, when the number of bits is less than 8 (the circuit area increases above 20%). When the number of bits becomes greater than 16, the difference in the circuit area decreases to less than 10%. For operands with more than 40 bits, this difference is approximately 2%, as depicted in Figure A.10, while improving the delay by more than 10%.

A.3.2 Binary and RNS multiplication units

In Figure A.11, it can be observed the relative efficiency of the multipliers by considering the product of the circuit area (A) by the square of the time (T) (AT^2). Given the quadratic variation of the area and the linear variation of the delay, the AT^2 metric is typically used as an efficiency metric [82, 103].

These results show a significant improvement, 20% in average and up to 50% for large ranges, regarding the Zimmermann architectures.

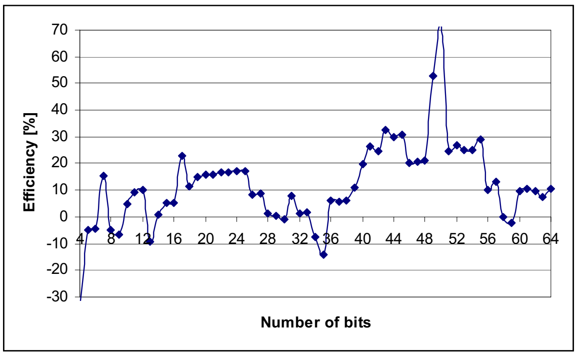


Figure A.11: Relative efficiency (AT^2) of the new multiplier regarding to the multiplier proposed by Zimmermann

However, the relative performance of the proposed multipliers is still lower than the one of the binary multipliers. Figure A.12 depicts the delay difference between the binary multiplier and the already optimized modulo $2^n + 1$ multiplier. This delay difference can rise to 70% (for $n = 4$), being on average about 20%.

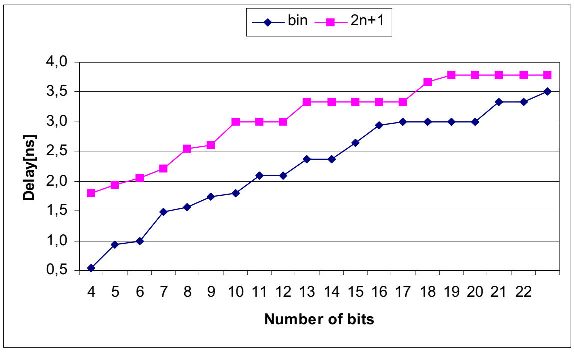


Figure A.12: Delay of the binary and the proposed modulo $2^n + 1$ multipliers

Also in the alternative diminished-1 number representation, proposed by Leibowitz in [104], for the residues modulo $2^n + 1$, the difference between the binary multipliers and the $2^n + 1$ multipliers is even higher, as depicted in Figure A.13. The recently published [82] optimized diminished-1 modulo $2^n + 1$

multiplier was used. In this analysis, The difference is in average about 30% faster.

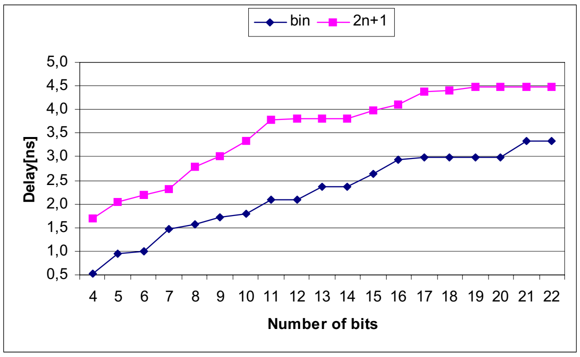


Figure A.13: Delay of the binary and the proposed modulo 2^n+1 diminished-1 multipliers

These results underline the need for a more balanced moduli sets, to compensate these significant differences in performance.

A.3.3 The new moduli set

The values presented for the new moduli set $\{2^n-1, 2^{n+k}, 2^n+1\}$ are for $0 \leq k \leq n$, while the values for the traditional moduli set are for $k = 0$. In Figure A.14, depicting the delay results for the binary to RNS converters, it can be seen that the converter for the new moduli set exhibits a slight performance improvement, about 4% on average. This is due to the shorter length of the modulo $(2^n \pm 1)$ adders used in the converters, since for the same dynamic range the value of n is smaller. However, these conversion units require about 16% more circuit area (Figure A.15) due to the usage of an extra CSA adder in the $2^n + 1$ channel to add the extra constant '1' (see Figure A.4 and equation (A.36)).

The proposed conversion units from binary to diminished-1 RNS are extremely compact, even more than the conversion units for the traditional moduli set. These conversion units are, on average, 10% faster (Figure A.16) and, unlike for the standard representation, require slightly less circuit area than the traditional moduli set (Figure A.17).

The RNS to binary converter for the new moduli set uses exactly the same structure as the converter for the original moduli set, with the advantage of

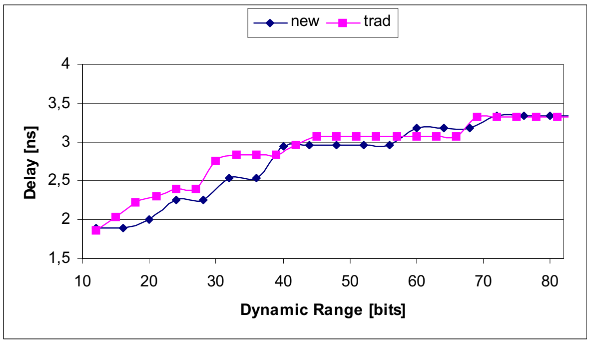


Figure A.14: Delay of the new and original binary to RNS converters

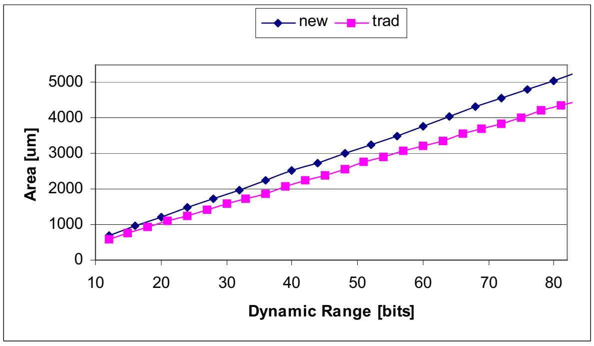


Figure A.15: Area of the new and original binary to RNS converters

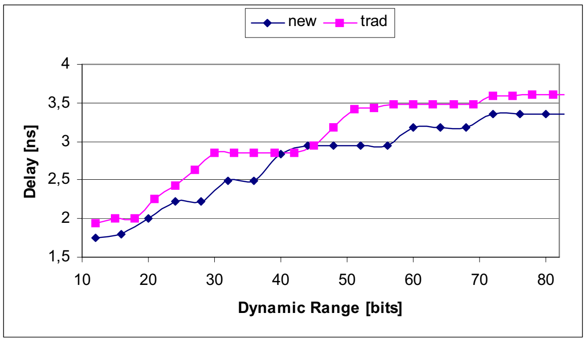


Figure A.16: Delay of the new and original binary to diminished-1 RNS converters

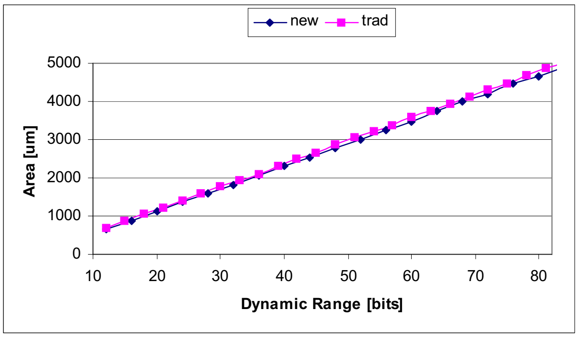


Figure A.17: Area of the new and original binary to diminished-1 RNS converters

having a smaller value of n, thus using smaller adders. This advantage not only causes an average reduction of about 10% in the conversion time (Figure A.18), but also a decrease of about 15% in the required implementation area (Figure A.19). Such improvement is mostly due to the longer binary channel (2^{n+k}), that is directly used to obtain the lower $n + k$ bits of the conversion result.

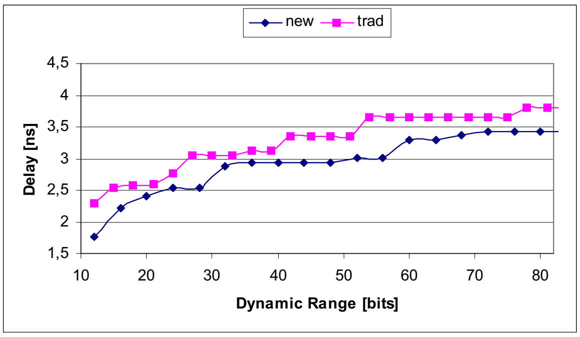


Figure A.18: Delay of the new and original binary to RNS converters

Taking into account the total cost of the two conversion units, the advantage of using the new moduli set becomes clear. Both the forward and the backward conversion units have approximately the same delay, which does not happen for the traditional moduli set, where the RNS to binary conversion is slower than the converter in the opposite direction. Concerning the required circuit area, the proposed new moduli set allows for a reduction of 2%, on average.

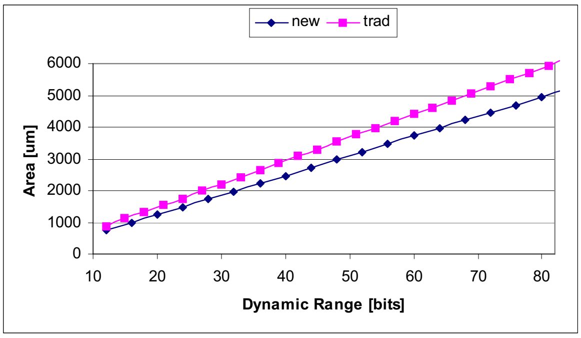


Figure A.19: Area of the new and original binary to RNS converters

A.3.4 ASIC RNS multiplication

The proposed moduli set achieves a more balanced processing delay between the arithmetic units due to the overloading of the binary channel, thus leading to an improvement of the overall computation efficiency. Figure A.20 depicts the delay difference for the multiplication operation, considering the new and the original moduli sets.

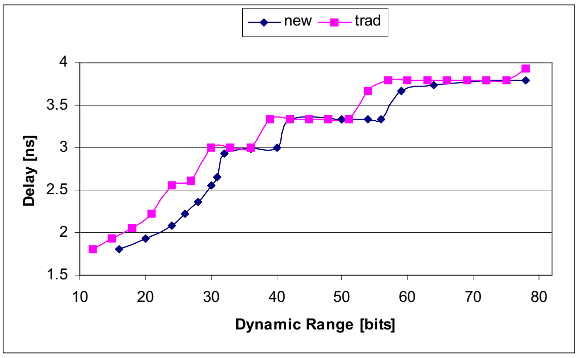


Figure A.20: Delay of the proposed multipliers for the new and original moduli sets

The use of the already optimized multiplier for the standard representation in the proposed moduli set allows an improvement in the multiplication delay up to 18%. For example, for $n = 17$ the proposed modulo $2^n + 1$ multiplier is 16% faster and 23% more efficient than the related art. Combining it with the proposed moduli set, for a dynamic range of 56 bits $(2^{17} - 1, 2^{22}, 2^{17} + 1)$, the

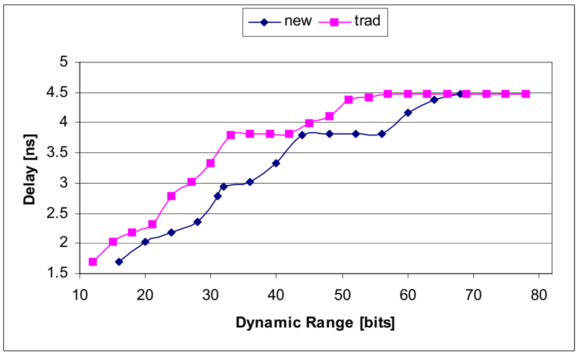


Figure A.21: Delay of the proposed diminished-1 multipliers for the new and original moduli sets

multiplication becomes even 12% faster. This results in an overall improvement in the multiplication time of 32%, regarding the case where neither the improved multiplier nor the proposed moduli set are used.

For the diminished-1 representation the improvement to the multiplication time is even higher. Figure A.21 depicts the difference between the multiplication delay in the original and in the proposed moduli set, using the already improved diminished-1 multiplier proposed by the authors of this paper [82]. For example, for a dynamic range of 56 bits the proposed moduli set allows to perform a multiplication 15% faster.

A.3.5 FPGA RNS multiplication

On technologies with embedded binary multipliers like in FPGAs, the use of RNS multiplication can also be significantly advantageous. As depicted in Figure A.22, for a Virtex II Pro FPGA, the performance of the binary embedded multiplier is significantly better than the non binary channels, up to 100% faster. It can also been seen that, even the improved $2^n + 1$ multiplication structure under performs the binary channel.

Figure A.22 also suggest a 24% improvement of the delay for the improved $2^n + 1$ multiplication structure with 32 bits, regarding the strait forward implementation of the $2^n + 1$ multiplication. This delay improvement is archived at a cost of less than 2% increase on the used slices and the same amount of embedded binary multipliers (252 Slices and 4 embedded multipliers).

Figures A.23 presents the performance values for the full multiplication struc-

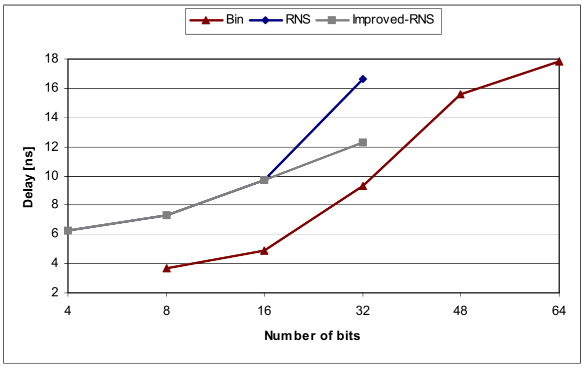


Figure A.22: Delay of the binary and moduli $2^n + 1$ multipliers

ture. From the delay trends, it becomes clear that the use of RNS to perform multiplications of large operands can be rather advantageous. Regarding the performance between the pure binary multiplier and the RNS multiplication, using the proposed balanced moduli set, improvements of approximately 100% can be achieved for operands with more than 64 bits. An Improvement of 21% can already be achieved for operands with 32 bits. The speed-up of the proposed moduli set multiplication tends to increase as the operands width increases.

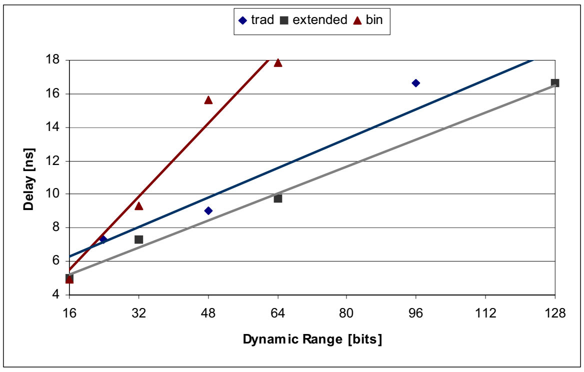


Figure A.23: Delay of the binary and the proposed RNS multiplication

Regarding the FPGA usage, for a 64 bit multiplier implemented purely in the binary representation, 10 embedded multipliers and 114 Slices are required. For the proposed RNS multiplication 5 embedded multipliers and approximately 225 Slices are required. This shows that a significant reduction in terms of hardware resource usage can also be achieved, along with a significant speed-up. Since in RNS smaller parallel multipliers are used, an even bigger area reduction is expected for larger operands, given that the multiplication area increases quadratically with the operands width.

A.4 Conclusions

In this thesis, an improvement to RNS systems that use the $2^n + 1$ modulo multiplication is proposed. The first step to improve the multiplication in RNS concerns the optimization of the multiplication unit itself. However, since the improved units are still slower than the binary multipliers, a new class of modified moduli sets that is capable of adapting the length of the binary channel ($\{2^n - 1, 2^{n+k}, 2^n + 1\}$) is also proposed.

The proposed ASIC multiplier is an improved architecture based on the modulo $2^n + 1$ multiplier proposed by Zimmermann. Experimental results suggest that multipliers 10% faster can be achieved, at the expense of some additional area. Nevertheless, for large dynamic ranges the area increase is not significant, requiring in average an additional 2% of silicon area. Furthermore, it can be concluded that for units using more than 9 bits, the proposed architecture is, on average, 20% more efficient than the original architecture.

The proposed extension of the binary channel allows the creation of more balanced and adaptable moduli sets. Conversion units were proposed for the particular case of the new 3-moduli set ($\{2^n - 1, 2^{n+k}, 2^n + 1\}$). Experimental results on ASIC, evidence more balanced forward and backward conversion times. Moreover, regarding the original conversion units, the proposed ones are faster and require less circuit area.

When both balanced moduli sets are adopted and enhanced multipliers are used, the overall multiplication can be further improved in up to 32%, with a delay reduction between 7% and 15%.

Implementations results suggest that, on technologies with embedded binary multipliers, performance improvements of 100%, for operands with at least 64 bits, can be achieved. The results also suggest that the performance of RNS multiplication tend to improve with the increase of the operands width.

Appendix B

Kitsos Whirlpool implementation

In Section 4.2.2 the Kitsos' [1] implementation of the Whirlpool algorithm is compared with the structure proposed in this paper. In this appendix, a more detailed analysis and comparison of the results presented in [1] is made.

One of the main computational differences between the structure proposed by us and the structure presented in [1], lays in the computation of the non-linear layer (γ) and the diffusion layer (θ). While in proposed structure, embedded memory blocks are used to implement both layers, in a coarse grained organization, in [1] a fine grained organization is used, where these two layers are implemented with Look Up Tables (LUTs) or Boolean expressions.

The non-linear computation performed in layer γ, is composed by 64 substitution tables (SBoxes), that in Kits [1] are implemented by the structure depicted in Figure B.1.

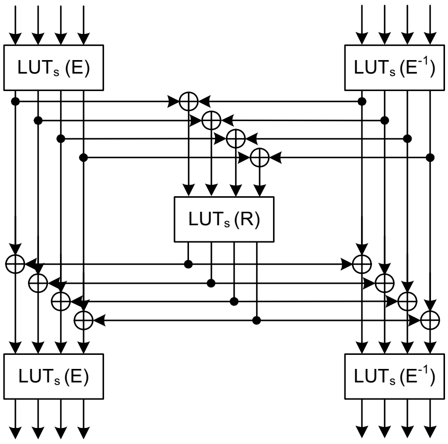


Figure B.1: Non-linear computation in Kits [1].

The LUT implementation of each of this SBoxes, can be easily estimated to have at least 3 LUTs in the critical path, without taking into account the routing delay. In terms of area, each of these SBoxes will require at least 4 LUTs for each E, E^{-1}, and R block, resulting in 20 (5×4) LUTs for each of the 64 SBoxes, giving a total of 1280 ($64 \times 5 \times 4$) LUTs for each γ layer.

For the diffusion layer θ, each of the 64 bytes (512 bits) has to be multiplied by the polynomial $g(x) = x$ modulo $(x^8 + x^4 + x^3 + x^2 + 1)$ in GF(2^8). The computation proposed by Kitsos' [1] is exemplified for bit b_{i0} in (B.1) and structurally depicted in Figure B.2.

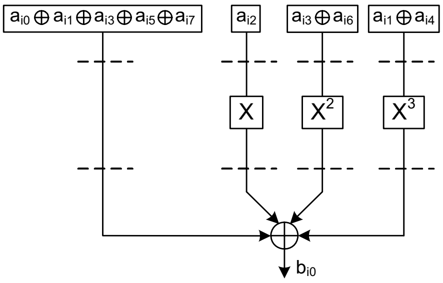


Figure B.2: Diffusion computation in Kits [1].

$$b_{i0} = a_{i0} \oplus a_{i1} \oplus a_{i3} \oplus a_{i5} \oplus a_{i7} \oplus X[a_{i2}] \oplus X^2[a_{i3} \oplus a_{i6}] \oplus X^3[a_{i1} \oplus a_{i4}] \tag{B.1}$$

The LUT implementation of each of these bit multiplications, can be estimated to have at least 2 LUTs in the critical path, without taking into account the routing delay. In terms of area, each of the 512 bits, will require at least 3 LUTs, resulting in a total area of at least 1536 (512 × 3) LUTs per θ layer.

Given that the compared structure uses one $w[\;]$ computation for the key expansion and another one for the Whirlpool core computation, at least 5632 (2 × [1280 + 1536]) LUTs are required just for the γ and θ layers, without considering the eventual area cost for routing. This value is larger than the synthesis figures presented by Kitsos' [1] for the whole Whirlpool computation, Table 4.6.

In terms of delay, the critical path for the γ and θ layers in [1] is of at least 5 (3 + 2) LUTs. Considering that the delay of a LUT is at least $1ns$ and the delay of a single BRAM is proximately $3ns$ [15], it is strange to find a maximum frequency 12% higher in Kitsos' [1] design, when compared with the proposed structure. Apart from the γ and θ layers, which in the proposed structure are implemented using a BRAM, the data paths of both designs are identical. Moreover, the proposed design is pipelined, which is not the case in [1].

Bibliography

[1] P. Kitsos and O. Koufopavlou, “Efficient architecture and hardware implementation of the Whirlpool hash function,” *IEEE Transactions on Consumer Electronics*, vol. 50, pp. 208 – 213, February 2004.

[2] Trusted Computing Group, “Main Specification Version 1.1b,” February 2002.

[3] Trusted Computing Group, “TCG Specification Architecture Overview,” April 2004.

[4] J.-C. Laprie, “Dependable Computing And Fault Tolerance :Concepts And Terminology,” vol. III, pp. 2–11, June 1996.

[5] B. Thuraisingham, “Dependable Computing For National Security: A Position Paper,” pp. 333 – 334, The Sixth International Symposium on Autonomous Decentralized Systems, ISADS, April 2003.

[6] A. Kulkarni, S.S.; Ebnenasir, “The Complexity Of Adding Failsafe Fault-Tolerance,” pp. 337 – 344, Proceedings. 22nd International Conference on Distributed Computing Systems, July 2002.

[7] Microsoft, “Next Generation Secure Computing Base,” 2003.

[8] Intel, “LaGrande Technology Architecture Overview,” September 2003.

[9] Intel, “Breakthrough Intel Tecnologies Move from the Descktop to Communications,” August 2005.

[10] Trusted Computing Group, “TPM Main Part 2 TPM Structures,” February 2005.

[11] Trusted Computing Group, “TPM v1.2 Specification Changes,” October 2003.

[12] NIST, “Announcing the advanced encryption standard (AES), FIPS 197,” tech. rep., National Institute of Standards and Technology, November 2001.

[13] R. Anderson, “Trusted computing’ frequently asked questions.” http://www.cl.cam.ac.uk/ rja14/tcpa-faq.html, August 2003.

[14] S. Vassiliadis, S. Wong, G. N. Gaydadjiev, K. Bertels, G. Kuzmanov, and E. M. Panainte, “The Molen polymorphic processor,” *IEEE Transactions on Computers*, pp. 1363– 1375, November 2004.

[15] Xilinx, *Virtex-II Pro and Virtex-II Pro X FPGA User Guide*, ug012 (v4.1) ed., March 2007.

[16] A. Seshadri, A. Perrig, L. van Doorn, and P. K. Khosla, “Swatt: Software-based attestation for embedded devices,” in *IEEE Symposium on Security and Privacy*, pp. 272–, IEEE Computer Society, May 2004.

[17] E. Shi, A. Perrig, and L. van Doorn, “Bind: A fine-grained attestation service for secure distributed systems,” in *IEEE Symposium on Security and Privacy*, May 2005.

[18] Douglas R. Stinson, “Cryptography - Theory and Practice,” CRC Press, 1995.

[19] A. J. Menezes, P. C. van Oorschot, and S. A. Vanstone, *Handbook of Applied Cryptography.* CRC Press, 2001.

[20] NIST, “Data encryption standard (DES), FIPS 46-3 ed,” tech. rep., National Institute of Standards and Technology, 1998.

[21] C. E. Shannon, “Communication theory of secrecy systems,” *Bell Systen Technicl Journal*, vol. 28, pp. 656–715, October 1949.

[22] NIST, “Data encryption standard (DES), FIPS 46-2 ed,” tech. rep., National Institute of Standards and Technology, December 1993.

[23] J. DAEMEN and V. RIJMEN, “The design of rijndael. AES-the advanced encryption standard,” *Springer-Verlag*, 2002.

[24] J.-C. Bajard and L. Imbert, “A Full RNS Implementation of RSA,” *IEEE Trans. Computers*, vol. 53, no. 6, pp. 769–774, 2004.

[25] H. Nozaki, M. Motoyama, A. Shimbo, and S. ichi Kawamura, "Implementation of RSA Algorithm Based on RNS Montgomery Multiplication," in Çetin Kaya Koç *et al.* [105], pp. 364–376.

[26] D. Mesquita, B. Badrignans, L. Torres, G. Sassattell, M. Robert, J.-C. Bajard, and F. G. Moraes, "A Leak Resistant Architecture Against Side Channel Attacks.," in *FPL*, August 2006.

[27] M. Neve, E. Peeters, and J.-J. Quisquater, "Parallel FPGA implementation of RSA with Residue Number Systems – Can side-channel threats be avoided?," *IEEE Midwest International Symposium on Circuits and Systems 2003 (Special session on Security and Cryptographic Hardware Implementations)*, December 2003.

[28] T. E. Gamal, "A public key cryptosystem and a signature scheme based on discrete logarithms," vol. 31, pp. 469–472, July 1985.

[29] V. Klima, "Finding MD5 collisions - a toy for a notebook." Cryptology ePrint Archive, Report 2005/075, 2005.

[30] X. Wang, Y. L. Yin, and H. Yu, "Finding Collisions in the Full SHA-1," in *CRYPTO* (V. Shoup, ed.), vol. 3621 of *Lecture Notes in Computer Science*, pp. 17–36, Springer, August 2005.

[31] NIST, "Announcing the standard for secure hash standard, FIPS 180," tech. rep., National Institute of Standards and Technology, May 1993.

[32] NIST, "FIPS 180-2, secure hash standard (SHS)," tech. rep., National Institute of Standards and Technology, August 2002.

[33] NIST, "The keyed-hash message authentication code (HMAC), FIPS 198," tech. rep., National Institute of Standards and Technology, March 2002.

[34] V. Rijmen and P. S. L. M. Barreto, "The WHIRLPOOL hash function." World-Wide Web document, 2001.

[35] FIPS, "Digital Signature Standard," FIPS PUB XX, February 1993.

[36] C. M. Wee, P. R. Sutton, and N. W. Bergmann, "An FPGA network architecture for accelerating 3DES - CBC," in *International Conference on Field Programmable Logic and Applications*, pp. 654–657, August 2005.

[37] G. Rouvroy, F.-X. Standaert, J.-J. Quisquater, and J.-D. Legat, “Design strategies and modified descriptions to optimize cipher FPGA implementations: fast and compact results for DES and triple-DES,” in *FPGA ’03: Proceedings of the 2003 ACM/SIGDA eleventh international symposium on Field programmable gate arrays*, (New York, NY, USA), pp. 247–247, ACM Press, 2003.

[38] CAST, “DES Cryptoprocessor Core – XILINX FPGA Results.” http://www.cast-inc.com/, 2007.

[39] P. Kocher, J. Jaffe, and B. Jun, “Introduction to differential power analysis and related attacks.” http://www.cryptography.com/dpa/technical, 1998.

[40] M.-L. Akkar and L. Goubin, “A generic protection against high-order differential power analysis.,” in *FSE* (T. Johansson, ed.), vol. 2887 of *Lecture Notes in Computer Science*, pp. 192–205, Springer, February 2003.

[41] L. Goubin and J. Patarin, “DES and differential power analysis (the ”duplication” method),” in *CHES ’99: Proceedings of the First International Workshop on Cryptographic Hardware and Embedded Systems*, (London, UK), pp. 158–172, Springer-Verlag, August 1999.

[42] M.-L. Akkar and C. Giraud, “An implementation of DES and AES, secure against some attacks,” in *CHES ’01: Proceedings of the Third International Workshop on Cryptographic Hardware and Embedded Systems*, (London, UK), pp. 309–318, Springer-Verlag, 2001.

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

[44] HELION, “High Performance AES (Rijndael) cores for Xilinx FPGA.” http://www.heliontech.com/, 2005.

[45] D. Kotturi, S.-M. Yoo, and J. Blizzard, “AES Crypto Chip Utilizing High-Speed Parallel Pipelined Architecture,” in *IEEE International Symposium on Circuits and Systems*, pp. 4653 – 4656, May 2005.

[46] S.-S. Wang and W.-S. Ni, “An efficient FPGA implementation of advanced encryption standard algorithm,” in *Proceedings of the 2004 International Symposium on Circuits and Systems*, vol. 2, pp. 597–600, May 2004.

[47] J.-F. Wang, S.-W. Chang, P.-C. Lin, and C. Kung, “A novel round function architecture for AES encryption/decryption utilizing look-up table,” in *IEEE 37th Annual 2003 International Carnahan Conference on Security Technology*, pp. 132– 136, October 2003.

[48] V. Fischer and M. Drutarovsky, “Two Methods of Rijndael Implementation in Reconfigurable Hardware,” in *Cryptographic Hardware and Embedded Systems, CHES 2001: Third International Workshop*, vol. 2162, pp. 77–92, January 2001.

[49] S. Morioka and A. Satoh, “A 10 Gbps Full-AES Crypto Design with a Twisted-BDD S-Box Architecture,” in *ICCD*, pp. 98–103, IEEE Computer Society, September 2002.

[50] B. Schneier, *Applied Cryptography: Protocols, Algorithms, and Source Code in C.* Jonh wiley & Sons, inc, second ed., 1996.

[51] CAST, “AES128-P Programmable Advanced Encryption Standard Core.” http://http://www.cast-inc.com/, 2005.

[52] Lemke, Schramm, and Paar, “DPA on n-bit sized boolean and arithmetic operations and its application to IDEA, RC6, and the HMAC-construction,” in *CHES: International Workshop on Cryptographic Hardware and Embedded Systems, CHES, LNCS*, pp. 205–219, August 2004.

[53] M. McLoone, C. McIvor, and A. Savage, “High-Speed Hardware Architectures of the Whirlpool Hash Function,” in *FPT* (G. J. Brebner, S. Chakraborty, and W.-F. Wong, eds.), pp. 147–162, IEEE, December 2005.

[54] R. Lien, T. Grembowski, and K. Gaj, “A 1 Gbit/s partially unrolled architecture of hash functions SHA-1 and SHA-512,” in *CT-RSA*, pp. 324–338, 2004.

[55] N. Sklavos, E. Alexopoulos, and O. G. Koufopavlou, “Networking data integrity: High speed architectures and hardware implementations,” *Int. Arab J. Inf. Technol.*, vol. 1, no. 0, 2003.

[56] L. Dadda, M. Macchetti, and J. Owen, “The Design of a High Speed ASIC Unit for the Hash Function SHA-256 (384, 512).,” in *DATE*, pp. 70–75, IEEE Computer Society, 2004.

[57] M. Macchetti and L. Dadda, “Quasi-Pipelined Hash Circuits,” in *IEEE Symposium on Computer Arithmetic*, pp. 222–229, IEEE Computer Society, 2005.

[58] T. Grembowski, R. Lien, K. Gaj, N. Nguyen, P. Bellows, J. Flidr, T. Lehman, and B. Schott, “Comparative analysis of the hardware implementations of hash functions SHA-1 and SHA-512.,” in *ISC* (A. H. Chan and V. D. Gligor, eds.), vol. 2433 of *Lecture Notes in Computer Science*, pp. 75–89, Springer, 2002.

[59] R. P. McEvoy, F. M. Crowe, C. C. Murphy, and W. P. Marnane, “Optimisation of the SHA-2 family of hash functions on FPGAs,” *IEEE Computer Society Annual Symposium on Emerging VLSI Technologies and Architectures (ISVLSI’06)*, pp. 317–322, 2006.

[60] N. Sklavos and O. Koufopavlou, “Implementation of the SHA-2 hash family standard using FPGAs,” *The Journal of Supercomputing*, vol. 31, pp. 227–248, 2005.

[61] M. McLoone and J. V. McCanny, “Efficient single-chip implementation of SHA-384 & SHA-512,” *proc. of IEEE International Conference on Field-Programmable Technology*, pp. 311–314, 2002.

[62] H. E. Michail, A. P. Kakarountas, G. N. Selimis, and C. E. Goutis, “Optimizing SHA-1 hash function for high throughput with a partial unrolling study,” in *PATMOS* (V. Paliouras, J. Vounckx, and D. Verkest, eds.), vol. 3728 of *Lecture Notes in Computer Science*, pp. 591–600, Springer, 2005.

[63] R. Chaves, G. Kuzmanov, L. A. Sousa, and S. Vassiliadis, “Rescheduling for optimized SHA-1 calculation,” in *SAMOS Workshop on Computer Systems Architectures Modelling and Simulation*, pp. 425–434, July 2006.

[64] R. Chaves, G. Kuzmanov, L. A. Sousa, and S. Vassiliadis, “Improving SHA-2 hardware implementations,” in *Workshop on Cryptographic Hardware and Embedded Systems, CHES 2006*, pp. 298–310, October 2006.

[65] CAST, "SHA-1 Secure Hash Algorithm Cryptoprocessor Core." http://http://www.cast-inc.com/, 2005.

[66] HELION, "Fast SHA-1 Hash Core for Xilinx FPGA." http://www.heliontech.com/, 2005.

[67] N. Sklavos and O. Koufopavlou, "On the hardware implementation of the SHA-2 (256,384,512) hash functions," *proc. of IEEE International symposium on Circuits and systems (ISCAS 2003)*, pp. 25–28, 2003.

[68] HELION, "Fast SHA-2 (256) hash core for xilinx FPGA." http://www.heliontech.com/, 2005.

[69] R. Chaves, G. Kuzmanov, L. Sousa, and S. Vassiliadis, "Merged Computation for Whirlpool Hashing," in *Proceedings of Design, Automation and Test in Europe 2008 (DATE 08)*, January 2008.

[70] R. Chaves, G. Kuzmanov, S. Vassiliadis, and L. Sousa, "Reconfigurable Memory Based AES Co-Processor," in *20th International Parallel and Distributed Processing Symposium (IPDPS 2006)*, IEEE, April 2006.

[71] N. Pramstaller, C. Rechberger, and V. Rijmen, "A compact FPGA implementation of the hash function Whirlpool," in *FPGA* (S. J. E. Wilton and A. DeHon, eds.), pp. 159–166, ACM, 2006.

[72] Xilinx, *Two Flows for Partial Reconfiguration: Module Based or Difference Based*, xapp290 (v1.2) ed., September 2004. Application Note: Virtex, Virtex-E, Virtex-II, Virtex-II Pro Families.

[73] Xilinx, *Virtex-5 Family Overview LX, LXT, and SXT Platforms*, ds100 (v3.2) ed., September 2007.

[74] S. Vassiliadis, S. Wong, and S. D. Cotofana, "The Molen $\rho\mu$-coded Processor," in *11th International Conference on Field-Programmable Logic and Applications (FPL), Springer-Verlag Lecture Notes in Computer Science (LNCS) Vol. 2147*, pp. 275–285, August 2001.

[75] J. Lu and J. Lockwood, "IPSec Implementation on Xilinx Virtex-II Pro FPGA and Its Application," in *Proceedings. 19th IEEE International Parallel and Distributed Processing Symposium*, pp. 158b – 158b, April 2005.

[76] Chodowiec, Gaj, Bellows, and Schott, "Experimental testing of the gigabit IPSec-compliant implementations of rijndael and triple DES using SLAAC-1V FPGA accelerator board," in *ISW: International Workshop on Information Security, LNCS*, pp. 220–234, 2001.

[77] P. Kocher, J. Jaffe, and B. Jun, "Introduction to differential power analysis and related attacks," 1998.

[78] N. Pramstaller, F. Gurkaynak, S. Hane, H. Kaeslin, N. Felber, and W. Fichtner, "Towards an AES crypto-chip resistant to differential power analysis," *ESSCIRC*, September 2004.

[79] R. Chaves, B. Donchev, G. Kuzmanov, L. Sousa, and S. Vassiliadis, "BRAM-LUT tradeoff on a Polymorphic DES Design," in *International Conference on High Performance Embedded Architectures & Compilers (HiPEAC 2008)*, January 2008.

[80] R. Chaves, G. Kuzmanov, L. Sousa, and S. Vassiliadis, "Cost-Efficient SHA Hardware Accelerators," *accepted for publication in IEEE Transactions on Very Large Scale Integration Systems (TVLSI).*

[81] R. Chaves and L. Sousa, "Faster Modulo $2^n + 1$ Multipliers without Booth Recoding," *XX Conference on Design of Circuits and Integrated Systems (DCIS)*, November 2005. ISBN: 972-99387-2-5 (full paper in CD-Rom format).

[82] L. Sousa and R. Chaves, "A universal architecture for designing efficient modulo 2^n+1 multipliers," *IEEE Transactions on Circuits and Systems-I: Regular Papers*, vol. 52, pp. 1166– 1178, June 2005.

[83] R. Chaves and L. Sousa, "$\{2^n+1, 2^{n+k}, 2^n-1\}$: A new RNS moduli set extension," *Symposium on Digital System Design (DSD 2004)*, pp. 210–217, September 2004.

[84] R. Chaves and L. Sousa, "Improving RNS multiplication with more balanced moduli sets and enhanced modular arithmetic structures," *Journal Computers & Digital Techniques, Institution of Engineering and Technology (IET)*, vol. 1, pp. 472–480, September 2007.

[85] M. Soderstrand, W. Jenkins, G. Jullien, and F. Taylor, eds., *Residue Number System Arithmetic: Modem Applications in Digital Signal Processing*. New York: IEEE, 1986.

[86] R. Chaves and L. Sousa, "RDSP: A RISC DSP based on residue number system," *Symposium on Digital System Design: Architectures, Methods and Tools*, pp. 128–135, September 2003.

[87] T. Toivonen and J. Heikkilä, "Video filtering with Fermat number theoretic transforms using residue number system," *IEEE Trans. Circuits Syst. Video Techn.*, vol. 16, no. 1, pp. 92–101, 2006.

[88] J.-C. Bajard and L. Imbert, "A Full RNS Implementation of RSA.," *IEEE Trans. Computers*, vol. 53, pp. 769–774, July 2004.

[89] H. Nozaki, M. Motoyama, A. Shimbo, and S.-I. Kawamura, "Implementation of RSA Algorithm Based on RNS Montgomery Multiplication.," in Çetin Kaya Koç *et al.* [105], pp. 364–376.

[90] X. Lai and J. L. Massey, "A proposal for a new block encryption standard.," in *EUROCRYPT*, pp. 389–404, 1990.

[91] S. Nakamura and K.-Y. Chu, "A single chip parallel multiplier by mos technology," *IEEE Transactions on Computers*, vol. 37, pp. 274–282, March 1988.

[92] E. D. Claudio, F. Piazza, and G. Orlandi, "Fast combinatorial RNS processors for DSP applications," *IEEE Transactions on Computers*, vol. 44, pp. 624–633, May 1995.

[93] M. Bhardwaj, T. Srikanthann, and C. Clarke, "A reverse converter for the 4-moduli superset $\{2^n-1, 2^n, 2^n+1, 2^{n+1}+1\}$," *Proceedings. 14th IEEE Symposium on Computer Arithmetic*, pp. 168 – 175, April 1999.

[94] R. Conway and J. Nelson, "Fast converter for 3 moduli RNS using new property of CRT," *IEEE Transactions on Computers*, vol. 48, pp. 852–859, August 1999.

[95] Virtual Silicon Technology Inc., *UMC High Density Standard Cells Library - 0.13μm CMOS process*, v2.3 ed., 1999.

[96] R. Zimmermann, "Efficient VLSI implementation of modulo $(2^n \pm 1)$ addition and multiplication," *Proc. IEEE Symp. on Computer Arithmetic*, pp. 158–167, April 1999.

[97] I. Koren, *Computer Arithmetic Algorithms*. Prentice-Hall, 1993.

[98] A.S.Ashur, M.K.Ibrahim, and A. Aggoun, “Novel RNS structures for the moduli set $(2^n - 1, 2^n, 2^n + 1)$ and their application to digital filter implementation,” *Signal Processing*, vol. 46, 1995.

[99] Y. Ma, “A slimplified architecture for modulo $(2^{\mathrm{n}} + 1)$ multiplication.,” *IEEE Trans. Computers*, vol. 47, no. 3, pp. 333–337, 1998.

[100] A. Hiasat and A. Sweidan, “Residue-to-binary decoder for an enhanced moduli set,” *IEE Proc.-Comput. Digit. Tech.*, vol. 151, pp. 127–130, March 2004.

[101] S. Andraos and H. Ahmad, “A new efficient memoryless residue to binary converter,” *IEEE Transactions on Circuits and Systems*, vol. 35, November 1988.

[102] S. J. Piestrak, “A high-speed realization of a residue to binary number system converter,” *IEEE Transactions on Circuits and Systems –II: Analog and Digital Signal Processing*, vol. 42, October 1995.

[103] E. D. D. Claudio, F. Piazza, and G. Orlandi, “Fast Combinatorial RNS Processors for DSP Applications.,” *IEEE Trans. Computers*, vol. 44, pp. 624–633, May 1995.

[104] L. Leibowitz, “A simplified binary arithmetic for the Fermat number transform,” *IEEE Trans. on Acoustics, Speech and Signal Processing*, vol. 24, pp. 356–359, October 1976.

[105] Çetin Kaya Koç, D. Naccache, and C. Paar, eds., *Cryptographic Hardware and Embedded Systems - CHES 2001, Third International Workshop, Paris, France, May 14-16, 2001, Proceedings*, vol. 2162 of *Lecture Notes in Computer Science*, Springer, 2001.

List of Publications

International Journals

1. R. Chaves, G. Kuzmanov, L. Sousa, and S. Vassiliadis “Cost-Efficient SHA Hardware Accelerators,” *IEEE Transactions on Very Large Scale Integration Systems (TVLSI).*
2. L. Sousa and R. Chaves, “A universal architecture for designing efficient modulo $2^n + 1$ multipliers,” *IEEE Transactions on Circuits and Systems-I: Regular Papers*, vol. 52, pp. 1166– 1178, June 2005.
3. R. Chaves and L. Sousa, “Improving RNS multiplication with more balanced moduli sets and enhanced modular arithmetic structures,” *Journal Computers & Digital Techniques, Institution of Engineering and Technology (IET)*, vol. 1, pp. 472–480, September 2007.

Conference Proceedings

1. R. Chaves, G. Kuzmanov, L. Sousa, and S. Vassiliadis, “Merged Computation for Whirlpool Hashing,” in *Proceedings of Design, Automation and Test in Europe 2008 (DATE 08)*, January 2008.
2. R. Chaves, B. Donchev, G. Kuzmanov, L. Sousa, and S. Vassiliadis, “BRAM-LUT tradeoff on a Polymorphic DES Design,” in *International Conference on High Performance Embedded Architectures & Compilers (HiPEAC 2008)*, January 2008.
3. R. Chaves, G. Kuzmanov, L. Sousa, and S. Vassiliadis, “Improving SHA-2 hardware implementations,” in *Workshop on Cryptographic*

Hardware and Embedded Systems, CHES 2006, pp. 298–310, October 2006.

4. R. Chaves, G. Kuzmanov, L. A. Sousa, and S. Vassiliadis, "Rescheduling for optimized SHA-1 calculation," in *SAMOS Workshop on Computer Systems Architectures Modelling and Simulation*, pp. 425–434, July 2006.

5. R. Chaves, G. Kuzmanov, S. Vassiliadis, and L. Sousa, "Reconfigurable Memory Based AES Co-Processor," in *20th International Parallel and Distributed Processing Symposium (IPDPS 2006)*, IEEE, April 2006. (full paper in CD-Rom format).

6. R. Chaves and L. Sousa, "$\{2^n+1, 2^{n+k}, 2^n-1\}$: A new RNS moduli set extension," *Symposium on Digital System Design (DSD 2004)*, pp. 210–217, September 2004.

7. R. Chaves and L. Sousa, "Faster Modulo $2^n + 1$ Multipliers without Booth Recoding," *XX Conference on Design of Circuits and Integrated Systems (DCIS)*, November 2005. ISBN: 972-99387-2-5 (full paper in CD-Rom format).

Curriculum Vitae

Ricardo Chaves was born on December the 25th, 1977, in Lisbon, Portugal. In 1996 he graduated from High School, with a technical degree in Electricity and Electronics. In 2001, Ricardo Chaves graduated from a 5 year university degree, in Electronics and Computer Engineering, from Instituto Superior Técnico (IST), Technical University of Lisbon (TULisbon). In 2003, after obtaining his MSc degree in Electronics and Computer Engineering at IST, Ricardo Chaves enrolled the Electronics and Computer Engineering Doctoral program at IST (TULisbon) under the supervision of Prof. Leonel Sousa, who had already been his advisor during both his undergraduate thesis and Master degree. During his MSc and first year of PhD studies, he worked as a research fellow in European projects, and also as a teaching assistant at IST. In the end of 2003 he was granted a 4 year PhD scholarship from the Portuguese Foundation for Science and Technology. In 2005, Ricardo Chaves visited the Computer Engineering (CE) lab of Delft University of Technology for 6 months as an exchange researcher within the European Network of Excellence High-Performance Embedded Architecture and Compilation; after these 6 months he decided to extend his PhD to a joint PhD program between IST, TULisbon and TUDelft. Prof. Stamatis Vassiliadis accepted to be his advisor whilst at TUDelft. His PhD work was focused on reconfigurable hardware structures for computational security.

The current research interests of Ricardo Chaves include: digital hardware, cryptography, and reconfigurable systems.