A Flexible Ultralight Hardware Security Module for EPC RFID Tags

by

Ahmed Ayoub

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Doctor of Philosophy
in
Electrical and Computer Engineering

Waterloo, Ontario, Canada, 2021

© Ahmed Ayoub 2021

Examining Committee Membership

The following served on the Examining Committee for this thesis. The decision of the Examining Committee is by majority vote.

External Examiner: Arash Reyhani-Masoleh
Professor, Dept. of Electrical and Computer Engineering,
Western University

Supervisor: Mark Aagaard
Associate Professor, Dept. of Electrical and Computer Engineering,
University of Waterloo

Internal Members: Guang Gong
Professor, Dept. of Electrical and Computer Engineering,
University of Waterloo

Hiren Patel
Professor, Dept. of Electrical and Computer Engineering,
University of Waterloo

Internal-External Member: Martin Karsten
Associate Professor, Cheriton School of Computer Science,
University of Waterloo

Author's Declaration

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

Due to the rapid growth of using Internet of Things (IoT) devices in daily life, the need to achieve an acceptable level of security and privacy for these devices is rising. Security risks may include privacy threats like gaining sensitive information from a device, and authentication problems from counterfeit or cloned devices. It is more challenging to add security features to extremely constrained devices, such as passive Electronic Product Code (EPC) Radio Frequency Identification (RFID) tags, compared to devices that have more computational and storage capabilities.

EPC RFID tags are simple and low-cost electronic circuits that are commonly used in supply chains, retail stores, and other applications to identify physical objects. Most tags today are simple “license plates” that just identify the object they are attached to and have minimal security. Due to the security risks of new applications, there is an important need to implement secure RFID tags. Examples of the security risks for these applications include unauthorized physical tracking and inventorying of tags. The current commercial RFID tag designs use specialised hardware circuits approach. This approach can achieve the lowest area and power consumption; however, it lacks flexibility.

This thesis presents an optimized application-specific instruction set architecture (ISA) for an ultralight Hardware Security Module (HSM). HSMs are computing devices that protect cryptographic keys and operations for a device. The HSM combines all security-related functions for passive RFID tag. The goal of this research is to demonstrate that using an application-specific instruction set processor (ASIP) architecture for ultralight HSMs provides benefits in terms of trade-offs between flexibility, extensibility, and efficiency. Our novel application specific instruction-set architecture allows flexibility on many design levels and achieves acceptable security level for passive EPC RFID tag. Our solution moves a major design effort from hardware to software, which largely reduces the final unit cost.

Our ASIP processor can be implemented with 4,662 gate equivalent units (GEs) for 65 nm CMOS technology excluding cryptographic units and memories. We integrated and analysed four cryptographic modules: AES and Simeck block ciphers, WG-5 stream cipher, and ACE authenticated encryption module. Our HSM achieves very good efficiencies for both block and stream ciphers. Specifically for the AES cipher, we improve over a previous programmable AES implementation result by 32×. We increase performance dramatically and increase/decrease area by 17.97/17.14% respectively. These results fulfill the requirements of extremely constrained devices and allow the inclusion of cryptographic units into the datapath of our ASIP processor.

Acknowledgements

First and foremost, I would like to praise and thank God who has granted countless blessing, knowledge, and opportunity to me so that I have been finally able to accomplish this thesis.

I would like to express my sincere appreciation to my supervisor, Professor Mark Aagaard for his tremendous support, guidance, encouragement and patience during my study at University of Waterloo. I have learned a lot from his vast experience and technical knowledge. Without his continuous support for my research and my personal life, this thesis would not have been possible.

I would like to express my appreciation to Professor Guang Gong for her support, and guidance. Many thanks for discussions, explanations, and clarifications during communications security lab (ComSec) seminars. I would like to thank my committee members, Professor Hiren Patel, Professor Martin Karsten, and Professor Arash Reyhani-Masoleh. For their time, good questions, and valuable comments.

I would like to give special thanks to all my colleagues and friends during my PhD studies. Special thanks to Nusa Zidaric, Gangqiang Yang, Bo Yang, Marat Sattarov, Jenny Yu, and all other past and present members of the ComSec lab.

Also, thanks to my former supervisors at Cairo University Professor Serag El-Din Habib, and Professor Hossam Fahmy who guided me through my BSc and MSc and taught me how to keep learning, be more productive and develop on the personal level.

I would like to express my gratitude to my parents Nagwa, and Abdelmordy. Without their encouragement, I would never reach to this point. All of my accomplishments through my life have been realized by endless love and support of them. Thanks to my sister Nahla and my brother Mohammad for their support and unconditional love.

Last but not least, I would like to express my deepest appreciation to my wife Hoda, my son Hamza, and my daughter Yasmine for their cooperation, sacrifices, and friendship during my PhD journey. Their love and friendship are great gifts in my life.

Dedication

This thesis is dedicated to my family, friends, and all the people who helped me to get this work done.

Table of Contents

List of Tables xii

List of Figures xiv

List of Abbreviations xvi

1 Introduction 1

2 Background 6

2.1 Security and Cryptography 6

2.1.1 Symmetric-key cryptography 10

2.1.1.1 Stream ciphers 10

2.1.1.2 Block ciphers 11

2.1.1.3 Mode of operations in block ciphers 12

2.1.1.4 Pseudorandom number generators 14

2.1.1.5 Hash functions 15

2.1.2 Lightweight cryptography 16

2.1.3 Physical security 18

2.1.4 Hardware security module 19

2.1.4.1 Secure Hardware Extension 20

2.1.4.2 Payment Card Industry Data Security Standard 20

2.1.4.3 Trusted Execution Environment 21
2.1.4.4 Trusted Platform Module 22
2.1.5 Security validations standards 22
2.1.5.1 FIPS 140-2 22
2.1.5.2 Common criteria 23
2.2 Digital Hardware Circuits 25
2.2.1 Digital hardware circuits design approaches 25

3 EPC RFID Systems 27
3.1 RFID standards 27
3.2 RFID system components and overall behaviour 29
3.2.1 Types of RFID tags 29
3.2.1.1 power supply source 30
3.2.1.2 Frequencies, Coupling method 31
3.2.1.3 Overall behaviour 33
3.3 EPC RFID tag specifications 35
3.3.1 RFID Tag reading range and Power 35
3.3.2 RFID Tag speed and response time 35
3.3.3 RFID tag microchip and memory banks 36
3.3.4 Reader stages/commands and Tag states: 37
3.3.4.1 Tag states: 38
3.3.4.2 Reader commands: 40
3.4 The EPC tag requirements 41
3.4.1 Functional Requirements: 42
3.4.2 Security Requirements: 42
3.4.3 Performance Requirements: 42
3.4.4 Area Requirements: 43
3.4.5 Power Requirements: 44

3.5 Related Work 44
3.5.1 Security for RFID tags 44
3.5.2 Hardware architectures for RFID tags 46
3.5.2.1 Software-based approach 46
3.5.2.2 Specialized hardware approach 47
3.5.2.3 ASIP-based approach 47

4 System level design **48**
4.1 Top-level design 48
4.2 The EPC tag digital baseband responsibilities 50
4.3 The interaction between the HSM and the CM 55

5 The HSM Architecture **59**
5.1 The overall architecture 59
5.2 Instruction-set architecture 61
5.3 Instruction Formats 62
5.3.1 Register Format (R-type) 62
5.3.2 First Immediate format (I1-type) 65
5.3.3 Second Immediate format (I2-type) 66
5.4 Instruction format design decisions 66
5.4.1 Three-operand vs. two-operand ISA architectures 67
5.4.2 Stack-based vs. register-based ISA architectures 67
5.4.3 Variable length instructions vs. fixed length instructions 68
5.5 The ASIP HSM features 68
5.5.1 Interrupt driven control 69
5.5.2 Input/output operations 70
5.5.3 Instruction-set extension 73
5.5.4 Accelerated mode support 78

5.5.5 Instruction fetch unit and control stack 78
5.5.6 Special-purpose registers 79
5.5.7 Memory management unit (MMU) 80

6 Results and Evaluations **83**
6.1 HSM architecture and results 83
6.2 Other work results 85
6.3 Comparison between our ASIP and Plos's and Groß's processor architectures 87
6.4 Cryptographic algorithms implementation results 92
6.4.1 Area considerations, ROM efficiency, and encryption/decryption efficiency 94
6.4.2 AES block cipher 96
6.4.3 Block ciphers: Present and Simeck 101
6.4.4 Stream ciphers: Trivium and WG-5 105
6.4.5 ACE authenticated encryption module 108

7 Conclusion and Future Work **113**
7.1 Conclusion 113
7.2 Future works 115
7.2.1 Implementation of more security services 115
7.2.2 Security validations 115
7.2.3 The communication module architecture 116

References **117**

APPENDICES **126**

A EPC commands **127**

A.1 NAK command 127
A.2 Query command 128
A.3 QueryRep command 129
A.4 QueryAdjust command 130
A.5 ReqRN command 131
A.6 Lock command 132
A.7 Select command 133
A.8 Read command 135
A.9 Write command 136

B Instruction-set architecture **137**

B.1 Memory/move instructions 137
B.2 Arithmetic instructions 138
B.3 Branch instructions 138
B.4 Output instructions 139
B.5 Custom instructions 139

List of Tables

2.1 The difference between the three hardware architectures 26

3.1 RFID operating frequencies and characteristic 32
3.2 T_1 timing and available clock cycles 43

4.1 Responsibilities in the communication module and the HSM 53
4.1 Responsibilities in the communication module and the HSM 54

5.1 The meaning and the encodings of the supported instructions 63
5.2 Examples of custom instructions encodings and meanings 64
5.3 Output instructions 73
5.4 Custom cryptographic instruction assembly code examples for different options 76
5.5 Branch/control instructions 79
5.6 Permission bits structure example 82

6.1 The chip area results for the ASIP design components 84
6.2 The related work implementation results 86
6.3 The chip area results 88
6.4 The selected cryptographic algorithms properties 93
6.5 Encryption/decryption operations efficencies for some cryptographic algorithms 95
6.6 AES Block encryption operation results in our ASIP HSM 97

6.7 Synthesis results of some AES algorithm implementations 98
6.8 AES Block encryption operation results in the related work 100
6.9 Synthesis results of some block ciphers implementations 101
6.10 Block encryption/decryption operation results 103
6.11 Encryption/decryption operation results using Simeck 32/64 on our ASIP HSM 105
6.12 Synthesis results of some stream ciphers implementations 106
6.13 Stream encryption/decryption operation results 107
6.14 Encryption/decryption operation results using WG-5 on our ASIP HSM . . 108
6.15 Loading/initialization execution time for ACE in our ASIP HSM 109
6.16 Encryption/decryption operation results for ACE in our ASIP HSM . . . 110
6.17 ACE instruction options in our ASIP HSM 111
6.18 Area results for ACE using our ASIP HSM 112

List of Figures

2.1 CIA triad 7
2.2 Symmetric-key cryptography system 7
2.3 Overview of cryptanalysis 9
2.4 General structure of the stream/block cipher 11
2.5 General structure of the stream cipher 11
2.6 One round function in block cipher 12
2.7 Block cipher structures 13
2.8 ECB encryption mode 13
2.9 CBC encryption mode 14
2.10 CTR encryption mode 15

3.1 RFID system components 30
3.2 Command-response communication protocol 34
3.3 Reader stages and tag states 38

4.1 Tag device components 49
4.2 Tag digital baseband architecture 52
4.3 ACK command operations division between the CM and the HSM 56
4.4 ACK command operations division between the CM and the HSM 58

5.1 The microarchitecture datapath for the ASIP HSM 60
5.2 R-type instruction format 65

5.3 I1-type instruction format 65
5.4 I2-type instruction format 66
5.5 The input/output interface signals 69
5.6 R_0 input register control logic 71
5.7 R_1 output register control logic 73
5.8 The input/output interface for a cryptomodule 74
5.9 The included Simeck cipher in the ASIP processor 75
5.10 The cryptomodule interface connections 77

6.1 Our ASIP processor area components 89
6.2 Plos's processor area components 90
6.3 HSM implementations using different cryptomodules 91

List of Abbreviations

AEAD	Authenticated Encryption with Associated Data
AES	Advanced Encryption Standard
ALU	Arithmetic Logic Unit
ASIP	Application Specific Instruction-set Processor
CBC	Cipher Block Chaining mode
CC	Common Criteria Standard
CM	Communication Module
Command	EPC standard command
CRC	Cyclic Redundant Check
CTR	Counter mode
DES	Data Encryption Standard
ECB	Electronic CodeBook mode
EPC	Electronic Product Code
EPCglobal	Electronics Product Code Global Incorporated
FIPS	Federal Information Processing Standards
FSM	Finite State Machine
GE	Gate Equivalent
GPP	General-Purpose Processor
HASH	Hash function
HF	High Frequency
HSM	Hardware Security Module
IEC	International Electrotechnical Commission
Instruction	What the user writes in assembly code
ISA	Instruction Set Architecture
IRQ	Interrupt Request
ISO	International Organization for Standardization
ISR	Interrupt Service Routine
IoT	Internet of Things

IV	Initialization Vector
LF	Low Frequency
NIST	National Institute of Standards and Technology
NRE	Non-recurring engineering
NVM	Non Volatile Memory
PCI-DSS	Payment Card Industry Data Security Standard
PRSG	Pseudo Random Sequence Generator
PRNG	Pseudo-Random Number Generator
RAM	Random Access Memory
RF	Radio Frequency
RFID	Radio-frequency identification
RISC	Reduced Instruction-set Computer
RN16	16-bit random number
ROM	Read Only Memory
SHA	Secure Hash Algorithm
sLiSCP	Simeck-based Permutations for Lightweight Sponge Cryptographic Primitives
SPN	Substitution-Permutation Network
SHE	Secure Hardware Extension
SOC	System-on-chip
TEE	Trusted Execution Environment
TID	Tag Identification
TPM	Trusted Platform Module
UHF	Ultra High Frequency

Chapter 1

Introduction

The Internet of Things (IoT) is a worldwide network that contains billions of interconnected physical devices. This network enables these devices to collect and exchange data. New applications require security and privacy with minimal possible cost. Radio Frequency Identification (RFID) is one of the technologies that is used to implement IoT applications.

RFID systems are widely used to perform automated identification for physical devices using radio frequency signals. Typically, an RFID system consists of three components: a tag, a reader and a back-end database. The tag is a microchip connected with an antenna, which can be attached to an object as the identifier of the object. The RFID reader communicates with the RFID tag using radio waves. The main advantage of RFID technology is the automated identification that promises changes across a wide range of business activities and aims to add new functionalities to systems that currently use bar codes.

One of the most popular RFID protocols is the EPC Class 1 Gen 2 standard. An Electronic Product Code (EPC) is a unique number that is stored in an EPC tag. Most tags today are simple "license plates" that just identify the object they are attached to, and have only minimal security. Due to the security risks of the new applications, there is an important need to implement cryptographically secure EPC tags. At the same time, adding these services is a significant challenge because of the strict cost, power consumption, and area limitations for the EPC tags.

This thesis presents an optimized application-specific instruction set processor (ASIP) for an ultralight Hardware Security Module (HSM). HSMs are physically separated computing devices that protect cryptographic keys and provide cryptographic operations for a device. We choose EPC tags as a prototype device. The HSM combines all security-related functions for an EPC tag. The goal of this research is to demonstrate that using an ASIP architecture for ultralight HSMs provides benefits in terms of trade-offs between flexibility, extensibility, and efficiency. Our ASIP processor can be implemented with 4,662 gate equivalent units (GEs) for 65 nm CMOS technology excluding cryptographic units and memories. We integrated and analysed four cryptographic modules: Simeck block cipher, WG-5 stream cipher, AES cipher, and ACE authenticated encryption module. We used existing implementations that can be easily plugged in our HSM. We increase performance dramatically and increase area by a small amount in comparison to previous works on adding cryptographic functions to EPC tags.

The main research questions for this thesis can be stated as follows: how feasible is using HSM design for such constrained devices such as EPC tags? And, how effective is using a flexible ASIP-based architecture for EPC tags? HSMs are typically implemented as extension cards or appliances for compute servers and other large, high systems. We have not seen a HSM design for constrained devices as EPC tags that have very limited hardware resources and power consumption. The other gap that we try to fill is adding high level of flexibility to EPC tags by using an ASIP architecture. This flexibility has many benefits in terms of simplicity of the design process to support different applications and to produce different tag models in one fabrication process.

Current EPC tag designs use two approaches: the specialised hardware approach, and the software-based approach. The specialised hardware approach can achieve the lowest area and power consumption and the highest performance for EPC tag designs, however it lacks flexibility and extensibility. The design process in this approach takes a long time and is relatively expensive. On the contrary, the designs that use the software-based approach are much slower and consume more energy. The main advantages of using software-based approach is the high flexibility and the relatively short time to market. The proposed ASIP-based approach can achieve a balance between rich functionality, low cost chips, high performance, and reasonable security requirements as well as high level of flexibility.

The ASIP-based design approach can be almost as efficient as the specialised hardware approach. The ASIP-based approach adds more hardware components and power consumption compared to the specialised hardware approach. However, ASIP processors

are designed for specific functions and include only the necessary hardware units. This results in low chip size and low power consumption, similar to that of specialised circuit designs. ASIP architecture allows flexibility and extensibility for EPC tags. In this thesis, flexibility is meant to be the ability to add cryptographic modules into the ASIP processor. The added cryptographic modules can be then accessed using custom instructions. Extensibility is the ability to write different software programs using same or different added cryptographic modules. In other words, extensibility is the ability to implement different tag models by writing different software programs.

Our proposed ASIP architecture for EPC tags has the potential to reduce the unit cost of tags with rich functionality. Compared to specialised hardware approach, the major part of the design effort in the ASIP-based approach is moved from the hardware (i.e. relatively expensive) to the software (cheap). This reduces the non-recurring cost (NRE). The ASIP-based approach allows large scale production by combining more than one tag model in one fabrication process. These tag models have same hardware components but they behave differently according to the included software programs.

The hardware development cost can be amortized over an extremely large number of tag units sold. Hence, the tag unit cost is reduced. The following example shows the potential effect of using the ASIP-based approach on the final unit cost. Suppose that the whole development cost is one million dollars. To get an amortized development cost of 0.1 dollars per tag, 10 millions tag units have to be produced. This does not include the manufacturing cost or profit, which also need to be included in the total price of the tag.

Today's license plate tags are simple; they don't have unique features for different applications. Many applications can use the same tag so it is possible to have demand for 10 millions or more license plate tags. Rich functionality leads to the need for different tag models (i.e. same hardware but different software programs) for different applications. It is possible to combine more than one tag model in one fabrication process to produce the same large number of tags using the ASIP-based approach.

The contributions of this thesis are listed as follows:

1. Introducing the idea and demonstrating the feasibility of creating an HSM that satisfies the constraints of RFID tags. Previous efforts to add security features to RFID tags have integrated the security functionality with the normal functionality. Our system architecture separates the HSM from the standard functionality.

2. Developing an ASIP architecture for an ultralight HSM that is significantly more *efficient* in performance vs. area than previous efforts to add security to RFID tags. Our HSM achieves very good efficiencies for both block and stream ciphers. Specifically for the AES cipher, we improve over a previous programmable AES implementation result by 32×.

3. Demonstrating that our ASIP architecture provides significant *flexibility* in plugging in a variety of cryptographic modules with minimal effort. Our ASIP instruction set provides customizable instruction for different ciphers and our micro-architecture provides a flexible but standardized interface to cryptographic modules. The cost of incorporating a cryptographic module into our HSM ranges from an area overhead of 2.37% and 40 lines of assembly code for the AES cipher to 10.39% overhead and 109 lines of code for the ACE authenticated cipher.

4. Demonstrating that our ASIP architecture provides significant *extensibility* in adding new functionality to the HSM. Adding support for authenticated encryption did not require any changes to the HSM instruction set or architecture.

5. In addition, the novel features of our HSM include:

    - Two registers in the register file are used for input/output operations, these registers allow for different serial/parallel loading modes.
    - The accelerated mode support; the ASIP can repeat the execution of one instruction multiple times without using a branch or loop instruction.
    - Implicit operations can be run after writing some value to the status register or to the accelerated mode/output counter register.
    - The integration of extended hardware units is done by using a generic interface. It may be required to design a wrapper for the added unit if its interface is not same as the used generic interface. The other traditional way for integrating a unit uses an intermediate standard bus to interact with the added unit.
    - An efficient instruction set optimized for HSMs. The instruction set doesn't include some common operations in most processors like multiplications and cryptographic operations. The main purpose of our ASIP processor is to move data between the memory, the input/output interface, and the extended hardware modules. Complex cryptographic computations are performed by the extended modules.

The rest of the thesis is organized as follows. Chapter 2 provides a background for the related topics and gives a review of the related works focused on secure implementations and lightweight security protocols for passive RFID tags. Chapter 3 discusses the system level architecture for the proposed tag design and explains the functions and the interface of each design's component. Chapter 4 provides a detailed description for the proposed ASIP HSM micro-architecture and its instruction-set architecture. Chapter 5 discusses the obtained results for our design and compares them to the other designs results followed by a conclusion and future works in Chapter 6.

Chapter 2

Background

2.1 Security and Cryptography

Security is the protection of the computer systems and its information from unauthorized access, disclosure, disruption, modification, inspection, recording or destruction. It is general term used with any form of data such as: electronic form, or physical form.

FIPS 199 Standard for Security Categorization of Federal Information and Information Systems by National Institute of Standards and Technology (NIST) [76] lists confidentiality, integrity, and availability as three security objectives for information and information systems, the three basic security objectives are known as the CIA triad as shown in Figure 2.1.

The FIPS-199 standard defines the confidentiality as preserving authorized restrictions on information access and disclosure, including means for protecting personal privacy and proprietary. In other words, a loss of confidentiality or privacy is the unauthorized disclosure of information. The integrity is defined as guarding against improper information modification or destruction, and includes ensuring information non-repudiation and authenticity. Availability is defined as ensuring timely and reliable access to and use of information.

The privacy and authentication of information are two security problems which cryptography studies and solves. The first term "privacy" can be achieved by encryption process which transforms a plaintext, the data in its original form, into a ciphertext. The function that transforms a plaintext message to a ciphertext message is called a cipher. The

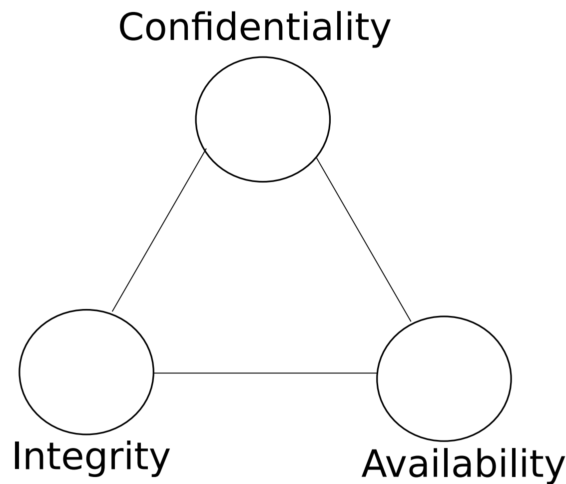


Figure 2.1: CIA triad

plaintext message afterwards could be recovered by the process of decryption. The second problem that cryptography solves is authentication. Message "authentication" means that the received message is actually sent by the true sender. Figure 2.2 shows a symmetric-key cryptography system that the cipher and the decipher has the same secret key.

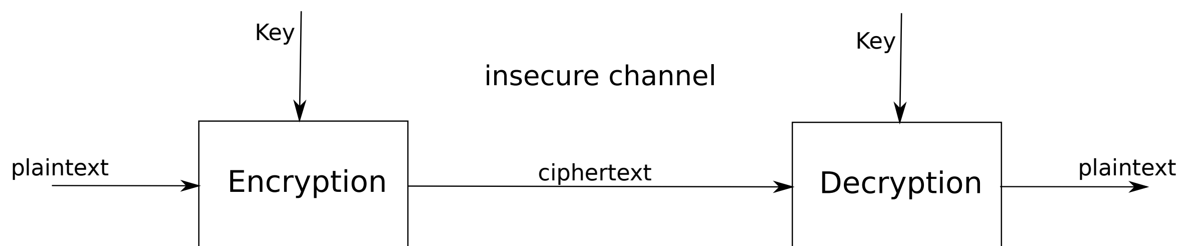


Figure 2.2: Symmetric-key cryptography system

The classical cryptanalysis is a reverse function of cryptography, which studies methods for obtaining the meaning of encrypted information, without access to the secret information such as the secret keys. The method that tries to break a crypto-system is called an attack. The attacks in the classical cryptanalysis are classified into four scenarios depending on the adversary capabilities. The adversary may eavesdrop the communications

among the entities, this attack is called *Ciphertext-Only Attack* (COA). For all wireless communication systems, the adversary has this capability to intercept communications among the different entities.

The second scenario is when the adversary has access to a limited number of pairs of plaintext and its corresponding ciphertext, this type is called *Known-Plaintext Attack* (KPA). The adversary has more opportunity to succeed by knowing more plaintext ciphertext pairs. In the third attack scenario (CPA), the adversary has the ability to generate a sequence of plaintexts of its choice and it has access to their corresponding ciphertexts. The last one is the most powerful model, called *Chosen-Ciphertext Attack* (CCA). In this attack model, the adversary can choose arbitrary ciphertext and have access to plaintext decrypted from it, it also can choose arbitrary plaintext and have access to the resulting ciphertext encrypted from it.

Another type of cryptanalysis relies on implementation attacks in which the attacker has physical access to a crypto-system such as smart cards, USB tokens, and RFID tags. This physical access may be gaining information from the implementation (i.e. side-channel attack), manipulating with the device to provoke an error (i.e. fault attack), or even spying on inner components of the chip (i.e probing attack).

The implementation attacks can be categorized according to their mechanical invasiveness or the attackers capabilities: non-invasive attacks, semi-invasive attacks, and invasive attack. Non-invasive attacks are considered usually as low-cost cryptanalysis which observe or manipulate with the device without physical harm. Semi-invasive attacks are more expensive than non-invasive attacks but the semiconductor chip is de-packaged but internal structure remains same as before. Invasive attacks are the most expensive method, the attacker has unlimited capabilities to extract information from chips and understand their functionality using expensive equipments. The different categories for cryptanalysis is shown in Figure 2.3. The focus of this thesis is on the hardware design for the HSM and demonstrating that the proposed ASIP architecture is more efficient than previous solutions. Security analysis is outside the scope of this thesis.

Modern cryptography is divided into two typical types: Symmetric-key cryptography and Public-key cryptography. In Symmetric-key cryptography, the sender and the receiver share the same secret key. This key is called symmetric key shared between the two parties. Public-key cryptography is a cryptographic system that uses two different keys; a public

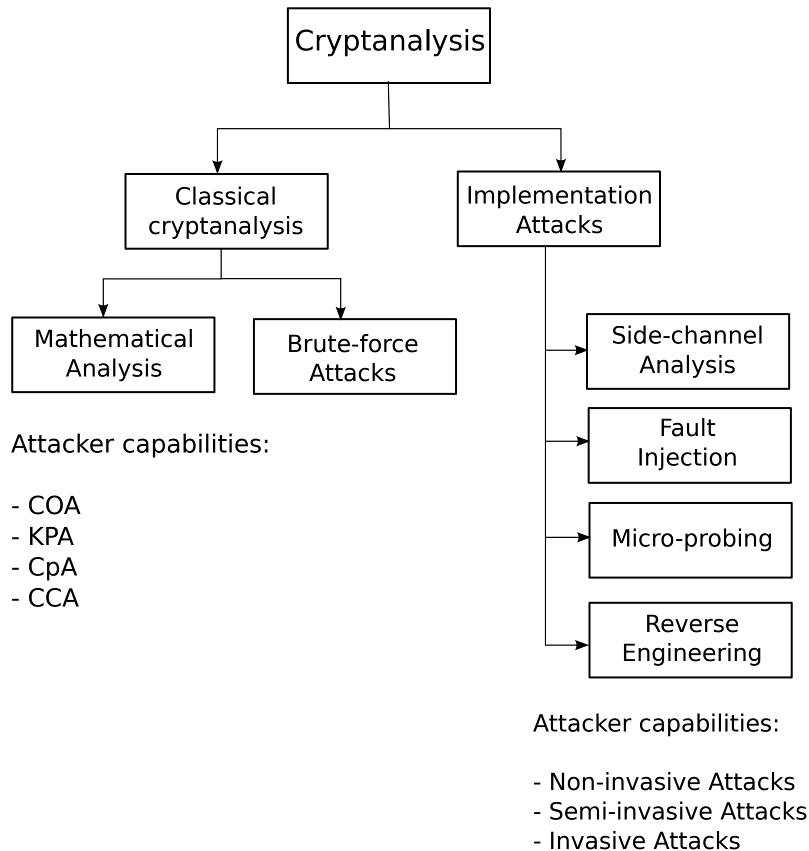


Figure 2.3: Overview of cryptanalysis

key and a private key. In case of encryption/decryption processes, a public key is used for encryption and a private key that is used for decryption. The public key is known to everybody but the private key is known only to the owner (as in the decryption entity) or to the owner and the third-trusted party that may generate it.

Both cryptography types have some advantages and disadvantages. The main advantage in symmetric-key cryptography is the speed and less complexity compared to public-key cryptography. The public-key cryptography involves intensive computations so it is not feasible for EPC tags due to area and power constraints. The key management in public-key cryptography is much easier compared to symmetric-key cryptography as the public key is known by everyone and could be shared within an insecure channel. The key in the symmetric-key cryptography should kept secret while transferring to any party.

One system may have many keys used for different cryptographic algorithms such as encryption/decryption key, and authentication/verification key. In symmetric-key cryptography, there are shared secret key for encryption/decryption algorithm and there is another shared secret authentication key for the authentication algorithm. On the other hand, a different keys pair is used for the authentication or in the digital signature algorithm, the public authentication key is used for signing while the private authentication key is used for the verification process. The public key could be shared and known by everyone.

In some systems, both cryptography types are used to combine the advantages from both types. Public key cryptography is used for the key establishment and symmetric-key exchange while symmetric key cryptography is used for the communications. In general, the symmetric key cryptography is used when performance is required as in the wireless communication systems.

This thesis uses only lightweight symmetric-key ciphers that are suitable for extremely constrained devices. Both block and stream ciphers have been used. AES is used in this thesis for benchmarking reasons to compare the efficiency to previous solutions. The generic kind of our solution allow for implementing different cryptographic mode of operations in software.

2.1.1 Symmetric-key cryptography

Symmetric-key ciphers are implemented as either block ciphers or stream ciphers. A block cipher encrypts input in blocks of plaintext, examples of the block lengths are 128-bit or 256-bit block lengths. A stream cipher enciphers instantaneously one-bit or a small number of bits such as 8-bit. Figure 2.4 shows a general structure of the stream/block cipher.

2.1.1.1 Stream ciphers

Stream ciphers is one kind of symmetric-key ciphers where a small number of plaintext bits is encrypted instantaneously (i.e. bit by bit or byte by byte). The general structure of the stream ciphers is shown in Figure 2.5. The keystream generator generates a keystream sequence depending on the seed or the input key. The key could be the initial state of the

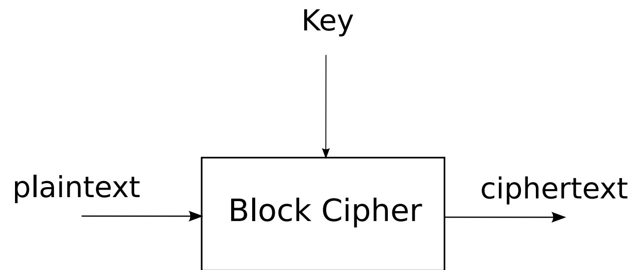


Figure 2.4: General structure of the stream/block cipher

feedback shift register, the input key is random. The ciphertext bit C_i is the XOR result between the input plaintext bit m_i and the keystream bit at a certain time i.

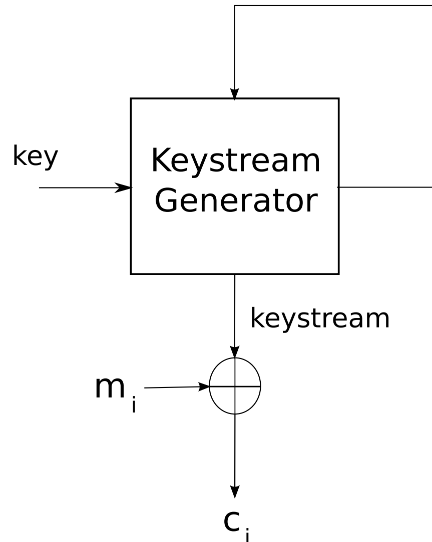


Figure 2.5: General structure of the stream cipher

2.1.1.2 Block ciphers

In general, the block cipher consists of two parts: round function and key schedule as shown in Figure 2.6. The round function is iterated multiple times in order to increase the unpredictability between the plaintext and the ciphertext. More rounds increase this unpredictability but it increases the total time to output the ciphertext as well. The key

schedule is used for generating new round key k_i for each round function iteration.

There are two common architectures (structures) for the round function in block ciphers: the substitution permutation network (SPN) and Feistel network.

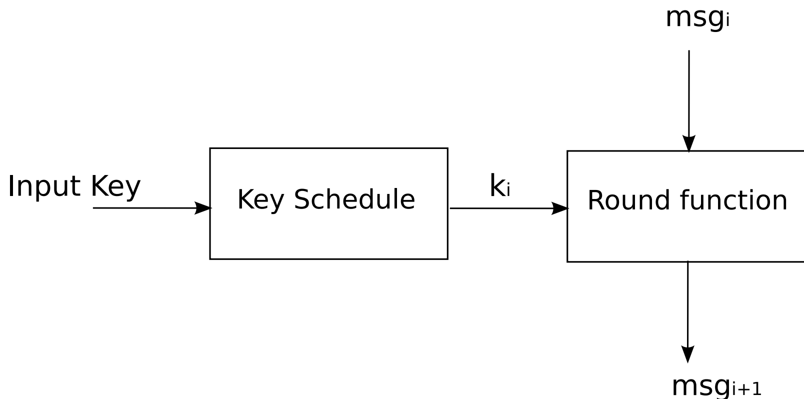


Figure 2.6: One round function in block cipher

The SPN structure consists of three typical layers: adding the round key, substitution layer, and permutation layer. The well-known example of SPN structure is Advanced Encryption scheme (AES)[22]. In the Feistel Network, the right half output comes directly from the left half input, the left half output is the XORed result of right half input and the output of the function **F** with inputs of the left half input and the Round-Key. One of the common block cipher that uses Feistel network structure is DES which is invented by IBM[33].

2.1.1.3 Mode of operations in block ciphers

The block ciphers have fixed input/output size, the cipher encrypt one block at time. In case of arbitrary length for the input message, block ciphers mode of operations describe how repeatedly to apply a ciphers's single block operation securely to transfer arbitrary length data larger than a block.

Many mode of operation for the block ciphers have been introduced. ECB or Electronic codebook mode is the simplest mode. In ECB mode, the message is divided into blocks

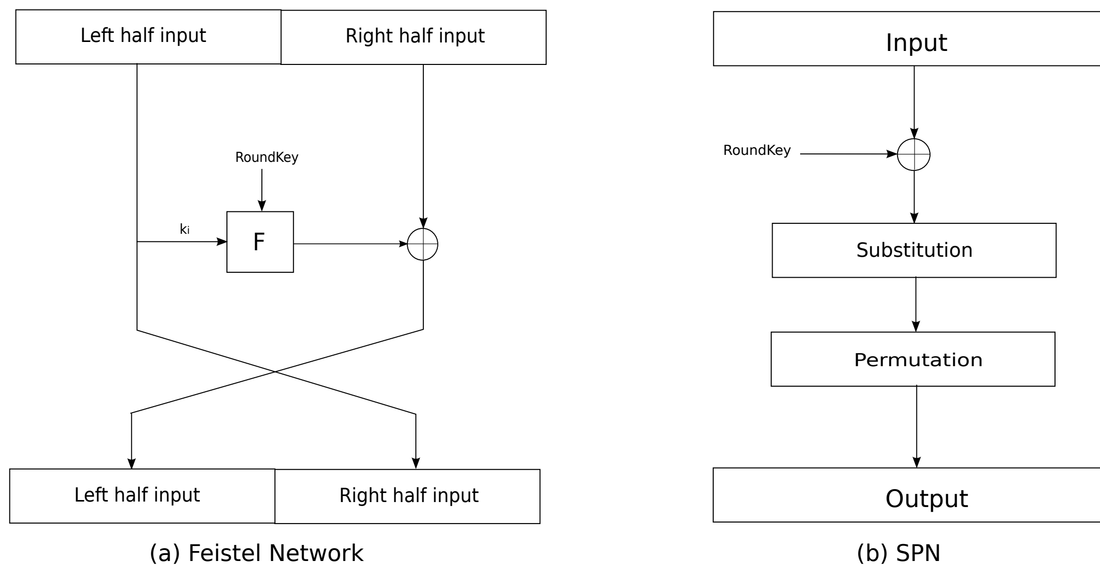


Figure 2.7: Block cipher structures

and each block is encrypted separately as shown in Figure 2.8. The main drawback of this mode that the repeated plaintext blocks are encrypted into repeated ciphertext blocks however it is easy to implement this mode of operation either in software or in hardware. This mode of operation is not secure under chosen plaintext attack (CPA).

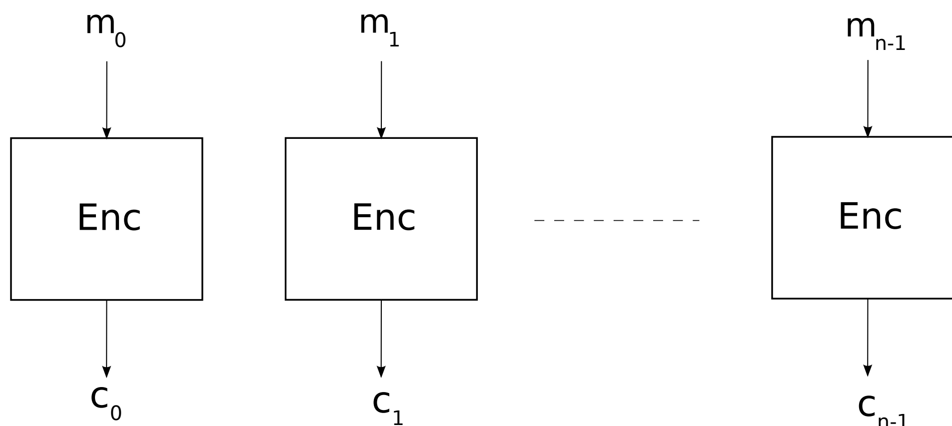


Figure 2.8: ECB encryption mode

Cipher Block Chaining (CBC) mode is another mode of operation that add some ran-

domization to the encryption algorithm. This leads to get different ciphertext blocks even the same plaintext blocks are repeated. This mode requires generating new random initial vector (IV) every new long message, this IV is sent in clear with the ciphertext blocks. It is most suitable to be implemented in software as every new plaintext message waits the previous ciphertext block to be XORed with it before being encrypted as shown in Figure 2.9.

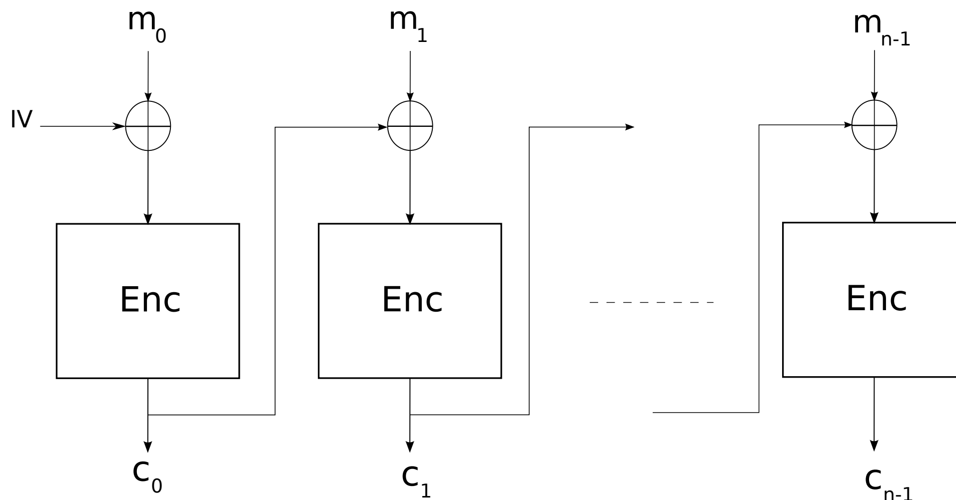


Figure 2.9: CBC encryption mode

The counter mode ((CTR) combined the benefits of ECB and CBC modes of operation, it generates the next keystream block by encrypting the successive values of a "counter". The "counter" is a sequence of incremented blocks that is initialized by *Nonce* random block. It is easy to be implemented in software or software, both encryption and decryption modes use the same encryption module (Enc) as shown in Figure 2.10. The *Nonce* is chosen randomly every new long message.

2.1.1.4 Pseudorandom number generators

Pseudorandom number generator (PRNG) is another important cryptographic module, the main function generates a sequence of random numbers. The generated sequence is not truly random as it is completely determined by the input key or the seed. This because that the function of PRNG is deterministic function. The generated sequence by the PRNG is

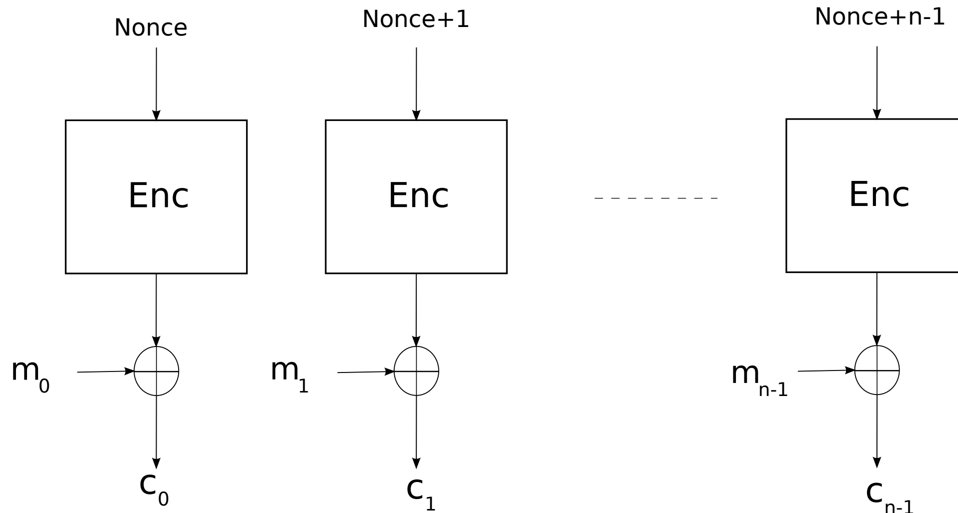


Figure 2.10: CTR encryption mode

distinguishable from a truly random number sequence.PRNGs are used to generate random numbers such as *Nonce* or *IV* or any required random number by a specific application. Also, PRNGs are used to generate keystream random sequence for the stream ciphers.

2.1.1.5 Hash functions

Hash function is another cryptographic module that is used to verify that a certain message maps to a specific hash value. In general, hash functions maps data of arbitrary size to a fixed size short bit string. The hash function is called one-way function as it is infeasible to invert.

Examples of hash function module include: MD5[82], SHA1[88], SHA2[38], SHA3[25]. Not all hash function modules are suitable for low-area low-power devices. Hence, there is a need to use lightweight hash modules, some lightweight hash function module examples include: Quark[11], PRESENT-based[17], PHOTON[47], Lightweight Keccak[57].

2.1.2 Lightweight cryptography

Lightweight cryptography is a cryptographic algorithm or protocol tailored for implementation in constrained environments including RFID tags, sensors, contactless smart cards. It targets systems that require a balance between security, cost, performance tradeoffs. In general, it is easy to optimize two of them but it is difficult to consider all of them [26].

In hardware implementations, chip size and energy consumption are two important measures that evaluate the lightweight properties. In software, the smaller code/RAM size is better for lightweight applications. ASIP design is divided between hardware and software, so all of these parameters are considered in our design with support of adequate security.

There are two main approach for implementing lightweight ciphers. The first approach develops hardware optimized implementations for standardized algorithms such as AES and DES block ciphers. The second approach is to design new ciphers suitable for lightweight cryptographic applications.

Several lightweight block cipher algorithms have been introduced such as PRESENT [16], CLEFIA [87]. Various compact hardware implementations of AES have been implemented in [65, 48, 36, 34]. A slight modification of DES cipher causes the appearance of lightweight DESXL cipher [74]. SIMON and SPECK [14] are two recent lightweight block ciphers.

Also some lightweight stream ciphers have been introduced as ECRYPT II eSTREAM project [83], Grain [50], Trivium [19], lightweight WG stream ciphers (WG-5[5], WG-7[62], WG-8[32]). Hummingbird [28] is another lightweight cipher that has a hybrid structure of block cipher and stream cipher.

Lightweight hash functions have been introduced such as Quark [11], PRESENT-based [17]. The standard hash functions such as MD5 and SHA-1 are not suitable for RFID tags due to their large hardware utilization.

PRESENT is an ultra-lightweight block cipher that achieves the required balance between security and hardware efficiency. The block size is 64-bit and the key size can be

80-bit or 128-bit. The non-linear layer is based on a single optimized 4-bit S-box. A problem with block collisions has appeared with block size 64-bit if they are used with large amounts of data [15]. Therefore implementations need to make sure that the amount of data encrypted with the same key is limited and re-keying is properly implemented.

Simon is a family of lightweight block ciphers released by the NSA. Simon has been optimized for performance in hardware implementations, while its sister algorithm, Speck, has been optimized for software implementations [14]. Simeck[91] combines the good design components from both SIMON and SPECK, in order to devise even more compact and efficient block ciphers. Simeck[91] has 10 instances depending on the input block size and number of key words. Simeck32/64, Simeck48/96, Simeck64/128 are some examples of instances have 4 key words and input block size 16, 24 , 32 respectively.

Simeck instances [91] are smaller than the similar ones of hardware-optimized cipher SIMON in terms of area and power consumption using same technology. Some security evaluations of Simeck have been done with respect to many traditional cryptanalysis methods, including differential attacks, linear attacks, impossible differential attacks, meet-in-the-middle attacks, and slide attacks. Overall, all of the instances of Simeck can satisfy the area, power, and throughput requirements in passive RFID tags.

Grain is a stream cipher submitted to eSTREAM project in 2004 [50]. Grain is designed primarily for restricted hardware environments. It accepts an 80-bit key and a 64-bit IV. The specifications do not recommended a maximum length of output per (key, IV) pair. A number of potential weaknesses in the cipher have been identified and corrected in Grain 128a which is now the recommended cipher to use for hardware environments providing both 128 bit security and authentication.

Trivium is a lightweight stream cipher designed to provide a flexible trade-off between speed and gate count in hardware, and reasonably efficient software implementation [19]. Trivium was submitted to the Profile II (hardware) of the eSTREAM competition, and has been selected as part of the portfolio for low area hardware ciphers by the eSTREAM project. It generates up to 264 bits of output from an 80-bit key and an 80-bit IV. The cipher itself consists of three NLFSRs. Trivium shows remarkable resistance to cryptanalysis for its simplicity and performance, recent attacks leave the security margin looking rather slim.

The WG stream cipher family [67] is fast stream ciphers. The original WG is a synchronous stream cipher submitted to the ECRYPT call. The general WG structure uses a word-oriented linear feedback shift register (LFSR) and a filter function based on the Welch-Gong (WG) transformation. Several instances of them have been explored in hardware, such as WG-29, WG-16, WG-7, and WG-5. The lightweight WG stream ciphers, WG-5, WG-7, and WG-8 have been proposed for the resource constrained environments. The lightweight WG ciphers can be used in protecting communication in these constrained devices, such as ensuring data confidentiality and performing entity authentication.

The WG cipher has been designed to produce keystream with guaranteed randomness properties such as: balance, long period, large and exact linear complexity, 3-level additive autocorrelation, and ideal 2-level multiplicative autocorrelation. It is resistant to Time/Memory/Data tradeoff attacks, algebraic attacks and correlation attacks [66].

2.1.3 Physical security

Physical security is a barrier to prevent unauthorized physical access to a device. Physical security should resist access, detect, respond, and/or provide evidence of tampering attempts at a later audit. A combination of tamper evidence, response or resistance can be used to create sufficiently strong level of protection to thwart many attacks.

In *tamper detection*, the device is designed so that it detects tamper attempts are happening. For example, an internal circuit can monitor extremes in the current or electrical properties to indicate a possible tamper event in the device. *Tamper response* refers to the defensive action taken by the device when tamper occurs. Tamper response typically takes actions such that the security assets in the system are not compromised. For example, making the device non-operational. *Tamper evidence* refers to auditing or logging the occurrence of a tamper event in the device. The device may have tamper log with more details which further actions can be taken. *Tamper resistance* is the ability of the device to detect and defend against a threat. The threat may be done by either normal users through normal interfaces or others with physical access to the device. For example, encryption of the sensitive information.

2.1.4 Hardware security module

Hardware Security Module (HSM) is a dedicated computing device that protects cryptographic keys and other security assets. It also provides all security services such as encryption/decryption, authentication, and digital signing services. Hardware Security Modules have to be validated for security by one of the security evaluations like FIPS 140-2 [77] or Common Criteria (CC)[1]. HSMs safeguard the cryptographic infrastructure by managing, generating, and storing cryptographic keys inside a protected device.

Including all secure assets in a separate device provides some benefits such as protecting the keys from memory scraping, and protecting the keys from physical theft. Beside these benefits, using hardware rather than software accelerates the cryptographic processing and insures the compliance with security validation evaluations. An HSM can have multiple levels of security and it can possess protection mechanisms to protect against tampering. In general, it works just like any other processing hardware but it is optimized for cryptographic algorithms and secured more thoroughly against physical security threats as well as logical security threats.

HSMs typically could be appliances, cards, or silicon chips. HSMs can be employed in any application that uses cryptographic keys and needs this type of physical protection. Examples of systems that use a HSM include: Card payment system HSMs; SSL, DNS, online banking; mobile payment and verbal banking; smart meters; medical devices; passports, identity cards, electronic passports; credit cards.

There are several specifications/standards in different industries for hardware protected security. The main goal for these specifications is to put a collection of characteristics of hardware mechanisms that are useful to certain industries and to their applications. For example, SAE J3101 [2] is a standard in automotive industry that puts some guidelines for implementing hardware security in ground vehicles. The guidelines cover hardware security related topics such as secure boot, secure storage, secure execution environment, access control, and authentication.

The same concept of including all secure assets in a separate zone has appeared in different names for various applications. Examples include:

- Secure Hardware Extension (SHE)

- Payment Card Industry Data Security Standard (PCI-DSS)
- Trusted Execution Environment (TEE)
- Trusted platform Module (TPM)

2.1.4.1 Secure Hardware Extension

Secure Hardware Extension (SHE) [31] is a specification that defines a set of functions that allows a secure zone (i.e. secure extension within a micro-controller unit) to exist within any electronic control unit installed in a vehicle. It was developed by Escrypt for Audi and BMW via the Hersteller Initiative Software (HIS) group. The secure zone features include storage and management of security keys, authentication, encryption and decryption algorithms that can access through software. Although the standard originated within the German automotive industry, it has since become an open standard accepted at the global level.

The E-safety Vehicle Intrusion proTected Applications (EVITA) has developed a set of guidelines for the design, and the verification of security architectures for automotive electronic control units. EVITA defines the overall functionality of three different hardware security module approaches: full, medium and light. Moreover, it specifies an elaborate set of functions and their parameters for managing security keys as well as encryption and decryption operations.

2.1.4.2 Payment Card Industry Data Security Standard

The Payment Card Industry Data Security Standard (PCI-DSS) [52] defines a set of logical and physical security compliance standards for HSMs specifically for the payments industry. It becomes a fundamental requirement for various payment processes, including PIN processing, card verification, card production, ATM interchange, cash-card reloading and key generation.

A payment platform must address some physical security and logical security requirements. Physical security requirements include:

- Tamper-detection and response mechanisms.

- Resilience to abnormal environmental and operating conditions.
- Protection of sensitive data within the device.
- Preventing disclosure of sensitive information by external monitoring techniques.
- Protection of cryptographic keys inside the device, even if the security boundary is breached.

To be PCI HSM compliant, A HSM software must address the following logical security requirements:

- Resilience against unexpected command sequences or operating modes.
- Secure firmware management.
- Strong authentication prior to running sensitive services.
- Secure key management and key separation to prevent misuse and eliminate exposure of sensitive data and PINs
- Secure audit trail

2.1.4.3 Trusted Execution Environment

Trusted Execution Environment (TEE) is typically offered to provide a secure hardware area in microprocessors. Usually it is built for commercial processors that supports complex operating systems as linux. It guarantees code and data loaded inside to be protected with respect to confidentiality and integrity. The Open Mobile Terminal Platform (OMTP) puts a set of defined threats that the TEE resists against[43]. It also puts two level of security; the first security level, Profile 1, was targeted against only software attacks and while Profile 2, was targeted against both software and hardware attacks. ARM TrustZone [3] TEE is an implementation of the TEE standard.

TEE retains its own hardware unique keys, stored in fuses. These keys are referred as "endorsement keys" or "provisioned secrets" which are embedded directly into the chip during manufacturing. The keys can be used to derive and secure other keys which can be used to uniquely identify the device.

2.1.4.4 Trusted Platform Module

Trusted Platform Module (TPM) is also known as ISO/IEC 11889 standard. It was specified by the Trusted Computing Group (TCG) and then it was standardized by International Organization for Standardization (ISO) and International Electrotechnical Commission (IEC) in 2009 [53] as ISO/IEC 11889, the current version is TPM 2.0[44].

TPM is a dedicated microprocessor designed to secure hardware through integrated cryptographic keys. TPM is usually used in PCs, laptops, mobile phones, and network equipment. Currently TPM is used by nearly all PC and notebook manufacturers.

The TPM allows for hardware-based cryptographic operations. Security functions can leverage the TPM for random number generation; the use of symmetric, asymmetric, and hashing algorithms; and secure storage of cryptographic keys and message digests. It ensures that no unintended users gain access to secret data by either stealing a device or via a software attack or brute force attack.

2.1.5 Security validations standards

For applications or devices that include cryptography, U.S. and Canadian federal government agencies are required to use a cryptographic products that has been FIPS 140 (Federal Information Processing Standards) validated or Common Criteria validated. Most Common Criteria protection profiles rely on FIPS validation for cryptographic security.

2.1.5.1 FIPS 140-2

FIPS 140 series are U.S. government computer security standards that specify requirements for cryptography modules. Within the FIPS 140-2 (or 140-1) validations, there are four possible security levels for which a product may receive validation:

1. Security Level 1 provides the lowest level of security. It specifies basic security requirements for a cryptographic module.
2. Security Level 2 improves the physical security of a Level 1 cryptographic module by adding the requirement for tamper evident coatings or seals, or for pick-resistant locks.

3. Security Level 3 requires enhanced physical security, attempting to prevent the intruder from gaining access to critical security parameters held within the module.
4. Security Level 4 makes the physical security requirements more stringent, and requires robustness against environmental attacks.

The security level of a particular device does not last forever. It is possible that a low cost attack will be found in the future when the attack tools become cheaper or available at second-hand.

FIPS 140 imposes requirements in different areas such as: cryptographic module specification, cryptographic module ports and interfaces, roles/services and authentication, finite state model, physical security, operational environment, cryptographic key management, electromagnetic interference/compatibility, self-tests, design assurance, and mitigation of other attacks. FIPS 140 standards define many tests that may be used to evaluate certain features in a system. For example, a particular part of FIPS 140-1 standard deals with statistical randomness tests such as: the monobit test, the pokers test, the runs test, and the long run test.

FIPS 140-3 [37] is a new version of the standard which is currently under development. In the first draft version of the FIPS 140-3 standard, NIST introduced a new software security section, one additional level of assurance (Level 5) and new Simple Power Analysis (SPA) and Differential Power Analysis (DPA) requirements.

Although the FIPS standards are developed by the US federal government, they are widely used as standards to specify the security requirements. Many companies acquire FIPS validation to comply with international standards. The FIPS validation will ensure that the proposed product or services meet the requirements standardized by a reputable organization and will make them competitive and open new markets.

2.1.5.2 Common criteria

The Common Criteria (CC) for Information Technology Security Evaluation is an international standard for computer security certification. The current version is 3.1 release 5. The computer system users, vendors, and testing laboratories participate in using Common Criteria by different ways. Computer system users can specify their security functional

and assurance requirements, vendors can then implement and/or make claims about the security attributes of their products and testing laboratories can evaluate the products to determine if they actually meet the claims. Common Criteria evaluations are performed on computer security products and systems, these systems may be software and/or hardware products.

The evaluation serves to validate claims made about the system. The evaluation must verify the system's security features. The claims are identified in some documents that are used as a part of the certification process. The protection profile (PP) is the document that is created by a user and provides an implementation independent specification of information assurance security requirements. Another document, that identifies the security properties of the system, is security target (ST) document. The CC 3.1 defines it as an implementation-specific statement of security needs for a specific identified system so it is provided by the vendor of the product. The individual security functions are specified in security functional requirements (SFRs) document.

The evaluation process also tries to establish the level of confidence through two quality assurance processes: security assurance requirements (SARs), and evaluation assurance level (EAL). SARs describe the measures taken during development and evaluation of the product to assure compliance with the claimed security functionality. The requirements may vary from one evaluation to the next. EAL has numerical rating that describes the depth of an evaluation. The increasing assurance levels reflect added assurance requirements that must be met to achieve Common Criteria certification. The intent of the higher levels is to provide higher confidence that the system's principal security features are reliably implemented. The EAL level does not measure the security of the system itself, it simply states at what level the system was tested.

The objectives of the Common Criteria agreement are to ensure that evaluations of IT products are performed to consistent standards and to improve the availability of security-enhanced IT products. It continuously improves, through the different versions, the efficiency and cost-effectiveness of the evaluation and certification/validation process for IT products.

2.2 Digital Hardware Circuits

2.2.1 Digital hardware circuits design approaches

In general, digital hardware circuits designs are divided into two basic approaches: the general-purpose approach, the special purpose approach. The general-purpose approach relies on hardware circuits like general-purpose processors or microprocessors that provide a fixed set of functionality (i.e. the instruction set of a microprocessor). On the other hand, the special-purpose approach involves design for a specific application. Specialised hardware circuits or application-specific instruction set processors are two hardware architectures that are used to design circuits in this approach.

The instruction set for general-purpose processor (GPP) must be general enough to support general applications. The general-purpose processor compiler should offer compilation for all programs. The main purpose of using general-purpose processors is getting the highest possible flexibility, they are designed to execute multiple applications and perform multiple tasks. So it is expensive especially for small devices that are designed to perform a specific task.

On the contrary, specialised hardware circuits provide the best performance, area, and power consumption but on the other side, they provide poor or no flexibility and extensibility. Any required modification needs the whole design and manufacturing process to start again, so the time-to-market is relatively high compared to GPP and ASIP-based designs.

In between of GPPs and specialised hardware circuits, there are application-specific instruction set processors. An application-specific instruction set processor (ASIP) is a processor which is customized for a certain application. It combines the benefits of the flexibility, and programmability as close to the general-purpose CPUs or the general-purpose processors (GPP) and the low silicon cost, low power consumption, and the performance as close to specialised circuits.

The hardware of ASIP and the instruction-set are designed together for one special application. ASIP designer thinks about the application and cost first. At the same time, ASIPs have reasonable flexibility. ASIP instruction-set is specifically designed to accelerate the most used functions for an application. Some operations that needs an extensive

Table 2.1: The difference between the three hardware architectures

Feature	General-purpose processor	Application-specific instruction-set processor	specialised hardware circuits
Flexibility	Excellent	Good	Poor
Power consumption	Relatively large	Medium	Small
Silicon Area	Relatively high	Low	Lowest
Performance	Depends on the application (mostly Low)	High	Highest
Energy Efficiency/speed	Low	High	Highest
Design Effort	System level	More on system level	Hardware
Hardware Design	Small	Large	Very large
Software Design	Large	Large	None
Time-to-market	Lowest	Relatively low	High
Cost	Mainly on Software	System-on-chip (SOC), Hw/Sw	Volume sensitive
Usage	Very wide	Wide	Limited

computation capabilities, as cryptographic functions, may need a dedicated hardware to perform it. ASIP allows adding dedicated hardware circuits which are accessed by special instructions. ASIP architecture is designed and optimized to implement the assembly instruction set with minimum hardware cost. The goal of an ASIP design is to reach high performance over silicon, over power consumption, as well over the design cost with reasonable level of flexibility. Table 2.1 summarizes the differences between the three digital hardware architectures.

Chapter 3

EPC RFID Systems

Radio-frequency identification (RFID) is a technology which the physical devices have an automated identification using Radio-frequency signals. In the last years, the RFID technology has found its way into our daily life such as supply chain management, home automation, e-cards, auto-mobiles, identification systems, ubiquitous and pervasive computing. The total RFID market was $10.1 billion in 2015 and it was expected to rise to $13.2 billion in 2020 [23]. Electronic Product Code (EPC) standards define the specifications of the tag and its communication protocols set by EPCglobal group. Different classes and generations were proposed, the last version is Class 1, Generation 2, Version 2.1 [40]. This chapter presents background information on Electronics Product Code (EPC) tags presented.

3.1 RFID standards

There are several RFID standards including ISO 18000 and EPCglobal standards. The standards are used to define the operations and the communication protocols between the RFID system components. These standards enable manufacturers to make products for large scale. It also enables products from different manufacturers to operate or work together.

There are two main organizations that make standards for RFID systems:

- ISO, International Standards Organization

- EPCglobal, Electronics Product Code Global Incorporated

The ISO/IEC 18000 series of standards deals with air interface protocol. The standards use five frequency bands for communication between the RFID system components. ISO/IEC 18000-2 standard uses 135 KHz for communication (low-frequency tags). It defines the air interface protocol, commands, and methods for detecting and communicating with one tag among several tags or so-called anti-collision. ISO/IEC 18000-3 standard uses frequency of 13.56 MHz for communication (high frequency tags). It defines same features in ISO/IEC 18000-2 standard but it adds definitions for physical layer, and collision management. ISO/IEC 18000-4 tags have frequency of 2.45 GHz for communication with the reader. The tags that conform this standard are mainly used for item management applications. It supports two mode of operations for tags: passive tags, or battery assisted tags.

ISO/IEC 18000-6 standard defines the communication protocol, commands between the reader and the tag, collision arbitration schemes for passive RFID system operating within 860-960 MHz frequency range (UHF). The standard describes three types A, B, and C. Type C is equivalent to EPCglobal Gen2 standard. ISO 14443 standard defines the operations for tags operating at 13.56 MHz using near-field inductive coupling. The tags in this standard are mainly used for identification cards. Typical applications include identity, security, payment, and access control. The read distance range is about 10 centimetres (3.94 inches). ISO 15693 standard has greater distance range compared to ISO 14443. The maximum read distance is 36 to 60 inches. This standard is used for controlling entry to a parking garage, airline baggage tracking, and supply chain management.

EPCglobal/GS1 Gen2 standard [29] was developed by EPCglobal, Inc. in 2004, which is now GS1, and was approved as ISO 18000-6C in July 2006. It defines air interface protocol for tags operating within the frequency range of 860–960 MHz. There are several new features compared to ISO standards:

1. Ability to change encoding according the noise in the environment.
2. Three modes for reader operations: single, multi, and dense environment.
3. Tag population management.
4. Longer kill and access passwords, 32-bit passwords.

5. Forward link data protection.
6. Four sessions for tag inventory.
7. Faster data transmission rate up to 640 Kbps.
8. Improved tag memory and programmability

3.2 RFID system components and overall behaviour

Typically, RFID systems consist of RFID devices or so called tags, RFID readers or interrogators, and backend database networks as shown in Figure 3.1. An RFID tag is a simple and low-cost electronic device that is attached to a physical object for wireless data transmission. It transmits data over the air in response to interrogation by an RFID reader. An RFID reader is a more powerful device (transceiver). Many readers can then connect to a network that acts as a data processing subsystem and database. Both reader and back-end server are powerful enough to handle the overhead introduced by performing strong cryptographic protocols. RFID tags usually have constrained capabilities in every aspect of computation, communication and storage due to the extremely low production cost and available power.

The reader starts the communication between the reader and the tag, and it follows a command/response pattern for communication, where the reader sends a command and the tag responds. As shown in Figure 3.2, each tag contains RF analog frontend, digital baseband. The RF part is used to perform two-way communication with the reader and harvest energy from the reader's signal. The digital baseband is used to process all the commands and data. The tag has up to four memories: EPC memory, TID (Tag Identification) memory, reserved memory, and user memory, the first three memories are mandatory while the user memory is optional.

3.2.1 Types of RFID tags

RFID tags can be classified based on many factors such as: power supply source, frequencies and coupling method,

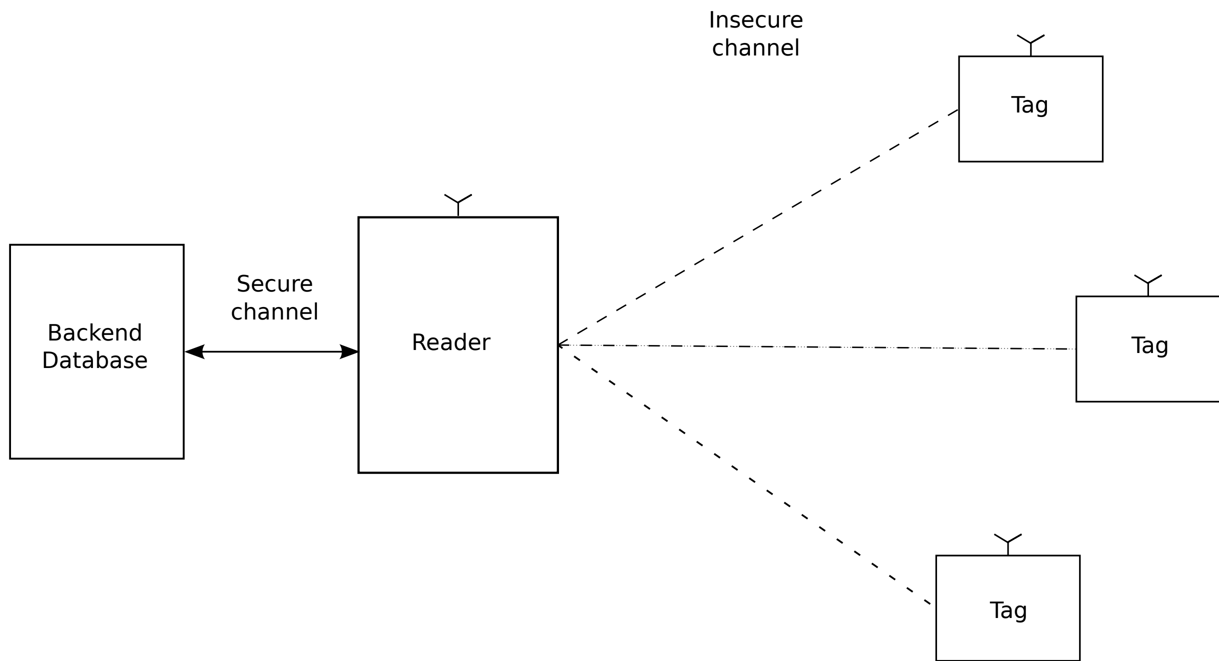


Figure 3.1: RFID system components

3.2.1.1 power supply source

Based on the tag's functionality, EPCglobal standard classifies RFID tags into four different classes:

1. Class 0 and 1:
   - Passive identity tags (usable range of 3 meters)
   - Backscatter (interrogator speaks first)
   - Lowest cost
2. Class 2:
   - Passive identity and memory tags (usable range of 3 meters)
   - Programmable
   - Backscatter (interrogator speaks first)
   - Security

- Lowest cost

3. Class 3:
   - Battery assisted passive tags
   - More functionality on chip – memory, sensors, etc
   - Backscatter (interrogator speaks first)
   - 100 meter range
   - Moderate cost

4. Class 4:
   - Active battery tags (tags transmit carrier)
   - Active transmission (permits tag-talks-first operating modes)
   - 100 meter range
   - High cost

Based on the powering method, an RFID tag may be passive (classes 0, 1, and 2), i.e. it does not have any power source of its own and derives its power from the RFID reader, active (class 4), i.e. it has its own power source (battery) and can initiate communication with reader, or semi-passive (class 3), i.e. it has its own power source but does not initiate communication with readers.

In passive tags, the received RF signal is rectified and the RF component is filtered to get DC voltage. The power control circuit in turn regulates and powers the circuits. Data from the reader is also extracted in the process using a demodulator and decoder. These tags use backscatter modulation to talk back to the reader. An active tag has a local power source, which also powers a transmitter. A semi-passive tag still uses the backscatter communication for talking back to the reader.

3.2.1.2 Frequencies, Coupling method

Wireless data transmission RFID systems use wide frequency range (125 KHz to 2.45 GHz). Table shows the main characteristics depending on the frequency band. The four main frequency bands are: low frequency (LF), high frequency (HF), ultra-high frequency (UHF),

Table 3.1: RFID operating frequencies and characteristic

Frequency band	LF	HF	UHF	Microwave
Frequencies	125-134 KHz	13.56 MHz	865-956 MHz	2.45 GHz
Reading range	up to 0.5 m	up to 1.5 m	up to 7 m	up to 10 m
Data transfer rate	Low	High	Medium	Medium
Coupling method	Near field	Near field	Far field	Far field
Applications	animal identification	contactless payment systems, ticketing, and library systems	toll collection, logistics, and supply-chain management	toll collection and baggage identification

and microwave. A main difference between LF and HF system to UHF and microwave systems is the way how the coupling between the reader and the tag is realised.

LF and HF systems use near-field coupling, where the wavelength is greater than the size of the tag antenna and the reading distance. The reader generates a magnetic field which induces an electric current in the tag's antenna. For the communication to the reader the tag changes the impedance of its antenna coil and the resulting change of current drawn can be detected by the reader. The maximum reading range of these systems is limited from a few cm up to at most 1.5 m. Power constraints to passive tags are lower and they have to deal with less interference from other readers or EM sources.

UHF and microwave systems operate in the far field where the wavelength is smaller than the average reading distance. Data transmission from tag to reader is done by changing the reflection coefficient of the tag antenna and the difference in reflection of the continuous EM waves is detected by the reader (backscatter). UHF systems allow higher data rates and read ranges but these systems have to cope with lower power supply of the tag, higher interference from other systems at similar frequencies and are more sensitive to reflection/disturbance from obstacles near the reader.

Operating ranges of RFID systems depend on various factors. Active tags do not need a reader RF field because of their independent power supply and usually enable higher

read ranges. Reading ranges from passive tags depend mainly on their power consumption and frequency range. Ultra-high frequency systems with far-field coupling enable greater operating ranges than inductive coupled systems that work in the near field only.

3.2.1.3 Overall behaviour

The EPC number has up to 496-bit size used for unique identification for the tag and it is stored in the EPC memory. The TID memory stores the unique tag identification number set by the manufacturer. Kill and access passwords are stored in the Reserved memory. The fourth memory is the user memory which is an optional memory for extending functionality of tags. Reader database contains the EPC numbers and the associated passwords of all tags.

According to the EPC C1 G2 standard, the reader has two states: inventory and access. The tag has five states: ready, reply, acknowledged, open, and secure as shown in Figure 3.2. The inventory and access protocol is executed as follows. The reader sends a Query command to the tag, and the tag replies with a 16-bit random number (RN16). After the reader receives the RN16, it sends an ACK command with the same RN16 to the tag. Then the tag sends the EPC number and Protocol Control (PC) bits, which comprise some protocol parameters such as EPC number length, to the reader if the RN16 it received is correct. Upon receiving it, the reader sends a ReqRN command with the same RN16 to the tag, the tag will reply with a new 16-bit random number called (Handle) to the reader after checking the correctness of the old RN16. After the reader receives the RN16 Handle, the reader may request the tag to execute one of the access commands with the previous Handle. Access commands include Read, Write, Kill, Lock and many other commands. The tag will execute the command and will reply with a new Handle after checking the correctness of the previous Handle.

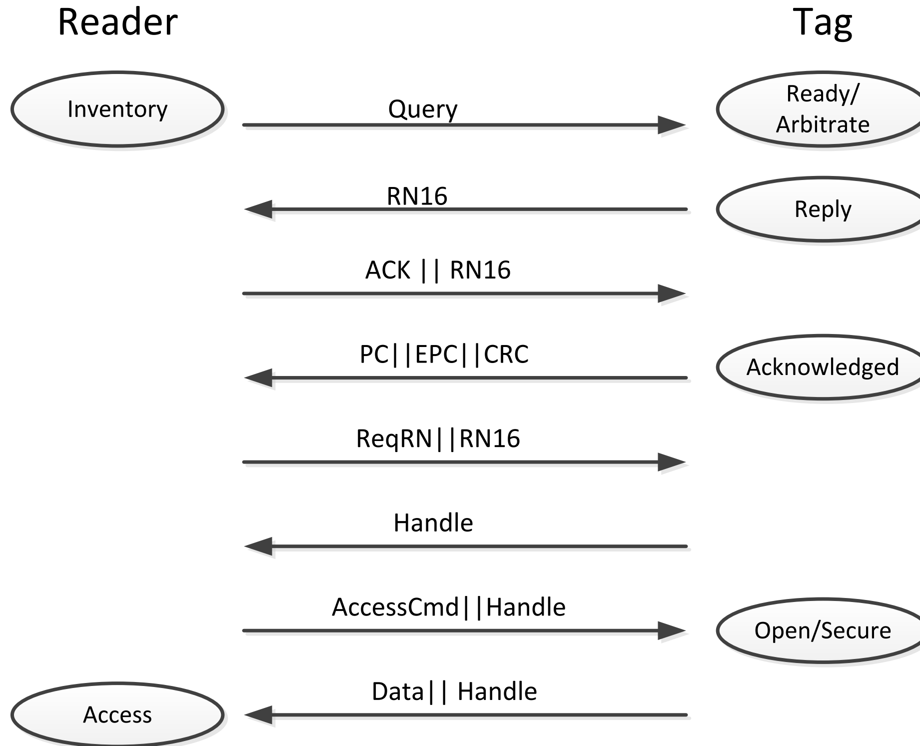


Figure 3.2: Command-response communication protocol

At the access command execution, the tag is either in Open state or in Secure state. Transferring from Open state to Secure state is done using ACCESS command after checking the correct 32-bit access password. Then, the tag ends up in the secure state and responds with the same Handle that the tag received. The reader can disable the tag forever using KILL command after checking the correct 32-bit kill password. KILL command transfers the tag to kill state, at this state the tag does not respond to the reader anymore.

Although the 32-bit access and kill passwords are two security features of the EPC tag device, they can be broken easily if the attacker has the ability to eavesdrop on the communication between the reader and the tag. As a result of that, the second version of the EPC C1 G2 standard adds some optional security requirements such as: Authenticate, SecureComm, AuthComm, KeyUpdate commands. AUTHENTICATE command allows the RFID to perform entity/mutual authentication between the reader and the tag. The

SECURECOMM allows a secure communication between the reader and the tag. These commands are optional commands that are specified by 8-bit Cryptographic Suite Indicator (CSI) using CHALLENGE or AUTHENTICATE commands.

3.3 EPC RFID tag specifications

An RFID tag is a wireless transmitter/receiver attached to a product. In general, it consists of a small microchip with an RF antenna, basic functional capabilities and limited memory. Each RFID tag includes an Electronic Product Code (EPC) which serves as a unique identifier (ID) for the physical object to which the tag is attached.

3.3.1 RFID Tag reading range and Power

EPC Class 1 Gen 2 RFID tags are low-cost passive write-many read-many tags that have a minimum memory of 256 bits, with possible extension to more tag memory. Ultra-High Frequency (UHF) Range is used for EPC tags. UHF range has frequencies from 860 MHz to 960 MHz which the read ranges of UHF systems are relatively large compared to High-Frequency and Low-Frequency distance ranges. Typical read ranges of UHF systems are 3 to 8 m. UHF is a good compromise between reading speed, distance, multiple tags handling and cost [12]. Important applications for UHF systems are toll collection, logistics, and supply-chain management.

The primary concern in passive RFID systems is the limited power that is available for the tags. Tags draw their energy from the electromagnetic field of a reader and use internal capacitors to buffer the energy to perform computations. The available energy depends thereby on various factors such as the distance to the reader, the size of the antenna, the operating frequency, and the field-strength of the reader.

3.3.2 RFID Tag speed and response time

Tags have to answer the reader within a specific response time. This time is usually very short, i.e., 15-250 us for EPC Gen2 tags (nominal range). The responses time are defined

in the standard and it depends on the Reply type the received command, other parameters in the standard. There are three tag reply response time options defined by the standard: Immediate, Delayed, and In-process response time.

The Reader-to-Tag bit rate is in the range of 26.7 kbps to 128 kbps assuming equiprobable data but the Tag-to-Reader bit rate is in the range of 5 kbps to 640 kbps.

3.3.3 RFID tag microchip and memory banks

The microchip consists of an analog front-end and a digital part. The analog front-end is responsible for demodulating the incoming data, extracting the clock signal, and modulating the response data. The analog front-end also extracts the power from the RF field. The digital part interprets the received data, performs the required actions (e.g. write data), and generates appropriate responses. Complexity of the digital part and functionality provided by it largely varies. At the upper end there are contactless smart cards, followed by sensor-enabled tags in the middle, and low-cost tags at the lower end.

RFID tag has four distinct memory banks which are known as tag memory. The memory banks are:

- **Reserved Memory** contains the kill and access passwords, the kill/access password size is 32 bits. If the kill/access passwords are not implemented, the default value is zero. Zero-valued kill/access passwords means that read/write operations are permanently locked.
- **EPC Memory** contains StoredCRC, StoredPC, EPC. The EPC code number identifies the object to which the tag is attached.
- **TID Memory** contains an 8-bit ISO/IEC 15963 allocation class identifier. It also contains some information for a reader to identify the custom commands and the optional features that a tag supports.
- **USER Memory** is optional memory that may be partitioned into one or more files, the file encoding should be defined either as in GS1 EPC Tag Data Standard or as in ISOIEC 15961 and 15962.

3.3.4 Reader stages/commands and Tag states:

Tag states are: Ready, Arbitrate, Reply, Acknowledged, Open, Secured, and Killed states. The defined commands by the standard and whether they are mandatory or not and their reply types are shown in the following table:

Command	Length (bits)	Mandatory (Y/N)?	Reply Type
QueryRep	4	Yes	Immediate
ACK	18	Yes	Immediate
Query	22	Yes	Immediate
QueryAdjust	9	Yes	Immediate
Select	> 44	Yes	None
NAK	8	Yes	Immediate
Req_RN	40	Yes	Immediate
Read	>57	Yes	Immediate
Write	>58	Yes	Delayed
Kill	59	Yes	Delayed
Lock	60	Yes	Delayed
Access	56	No	Immediate
BlockWrite	> 57	No	Delayed
BlockErase	> 57	No	Delayed
BlockPermalock	> 66	No	Immediate & Delayed
ReadBuffer	67	No	Immediate
FileOpen	52	No	Immediate
Challenge	> 48	No	None
Authenticate	> 64	No	In-process
SecureComm	> 56	No	In-process
AuthComm	> 42	No	In-process
Untraceable	62	No	Delayed
FileList	71	No	In-process
KeyUpdate	> 72	No	In-process
TagPrivilege	78	No	In-process
FilePrivilege	68	No	In-process
FileSetup	71	No	In-process

On the other side, the reader manages tag population using three stages: Select, Inventory, and Access stages. In Select stage, the reader chooses a tag population, all tags

in close to the reader are involved in this stage. The reader may use *Select* command to select one or more tags based on value in tag memory. *Challenge* command may be used to challenge one or more tags based on the supported cryptographic suite and authentication type by the tag. This stage comprises *Select* and *Challenge* commands.

In Inventory stage, the reader identify individual tags by transmitting a *Query* command. One or more tags may reply, the reader detects a single tag reply then requests the tag's EPC number. The third stage is starting the communicating with as identified tag, this stage is called Access stage. The reader may read, write, lock, or kill the tag. All security-related operations such as authenticating the tag are involved in this stage. Figure 3.3 shows the different reader stages and the tag states.

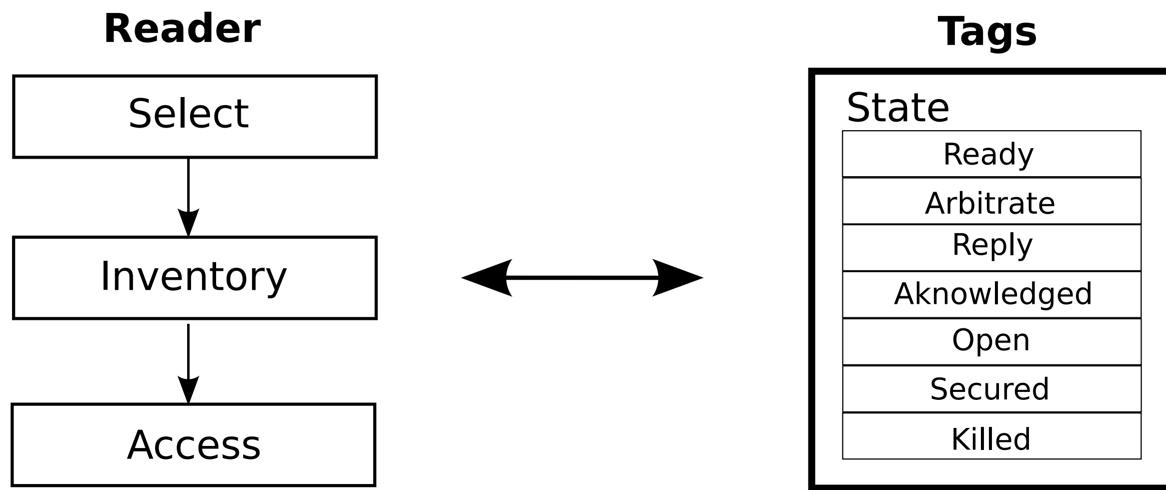


Figure 3.3: Reader stages and tag states

3.3.4.1 Tag states:

The tag states are described as following:

1. **Ready state:** Ready state can be viewed as a "holding state" for energized tags that are neither killed nor currently participating in an inventory round. Upon entering an energizing RF field a tag that is not killed shall enter ready.

If the tag receives a *Query* command matching parameters as specified in the standard, the tag changes the state to Arbitrate state if the slot counter is non-zero or to Ready state if the slot counter is zero.

2. **Arbitrate state:** can be viewed as a "holding state" for tags that are participating in the current inventory round but whose slot counters hold non-zero values.

   Each time the tag receives a *QueryRep* command with matching parameters as specified in the standard, the tag decrements the slot counter. The tag transitions to Reply state when its slot counter reaches zero.

3. **Reply state:** Upon entering reply a tag shall backscatter an RN16. If the tag receives a valid acknowledgement (*ACK* command) then it shall transition to the Acknowledged state.

4. **Acknowledged state:** A tag re-backscatter a reply depending on the flags values stored in a tag as stated in the standard. In general, the reply consists of these parameter fields: PC/XPC, full/truncated EPC, and calculated CRC for the first two fields. A tag in Acknowledged state may transition to any state except Killed state depending on the received command. For example, if a tag in the Acknowledged state receives a valid *ACK* command containing the correct RN16 then it shall re-backscatter the reply and it stays in the same Acknowledged state. If a tag in the Acknowledged state fails to receive a valid command within a specific time $T_2(max)$ then it shall return to Arbitrate state.

5. **Open state:** Open state could be viewed as a non-secure Read/write state. It allows executing some access commands as: *Read*, *Write*, *Lock* commands. A tag in Acknowledged state transitions to Open state when it receives an *Req_RN* command with correct RN16 and correct non-zero access password.

6. **Secured state:** Secured state could be viewed as a secure Read/write state. It allows executing some access commands as: *Read*, *Write*, *Lock* commands after receiving a correct access password with *Access* command.

7. **Killed state:** Kill state permanently disables a tag. Upon entering the Killed state, a tag shall notify the reader that the kill was successful and shall not respond to a reader thereafter. Killed tags shall remain in the killed state under all circumstances, and shall immediately enter Killed state upon subsequent power-ups. Killing a tag is irreversible. A tag is entered the Killed state upon receiving either a successful password-based Kill sequence or a successful authenticated Kill sequence.

3.3.4.2 Reader commands:

Some important reader's commands are described as following:

- Select stage commands:
  - ***Select*** command allows a reader to select a particular tag population prior to inventorying. Upon receiving a Select a not-killed tag returns to the ready state, evaluates the criteria, and depending on the evaluation may modify the indicated SL or inventoried flag.
  - ***Challenge*** command allows a reader to instruct multiple tags to simultaneously yet independently pre-compute and store a cryptographic value or values for use in a subsequent authentication. The generic nature of the Challenge command allows it to support a wide variety of cryptographic suites.
- Inventory stage commands:
  - ***Query*** command initiates and specifies an inventory round. Query command contains a slot-count parameter Q. Upon receiving a Query participating tags pick a random value in the Q range (0, 2^{Q-1}), inclusive, and load this value into their slot counter.
  - ***QueryAdjust*** command adjusts Q (i.e. the number of slots in an inventory round) without changing any other round parameters.
  - ***QueryRep*** instructs tags to decrement their slot counters and, if slot=0 after decrementing, to backscatter an RN16 to the reader.
  - ***ACK*** command may be issued by a reader to cause a single tag to backscatter its EPC. Adter correct ACK command (i.e. correct RN16), the tag replies by the reply that consists of these parameter fields: PC/XPC, full/truncated EPC, and calculated CRC as specified in the standard.
  - ***NAK:*** A tag that receives a NAK shall return to the Arbitrate state without changing its inventoried flag, unless the tag is in Ready or Killed states.
- Access stage commands:
  - ***Req_RN*** instructs a tag to backscatter a new RN16.
  - ***Read*** allows a reader to read part or all of a tag's Reserved memory, EPC memory, TID memory, or the currently open file in User memory.

- ***Write*** command writes 16 bits (one word) at a time, using link cover-coding to obscure the data during Reader-to-Tag transmission. A *Write* command does not alter the tag state.
- ***Kill*** command allows a reader to permanently disable a tag using one of the kill procedures (password-based kill sequence, optional authenticated-kill sequence).
- ***Lock*** command may be issued to lock/unlock the kill password, access password, EPC memory bank, TID memory bank, or File_0 of user memory. If the passwords are locked or permanently locked then they are unwriteable and unreadable by any command and usable only by a Kill or Access command. If EPC memory, TID memory, or File_0 are locked or permanently locked then they are unwriteable but readable, except for the L and U bits in EPC memory.
- ***Access*** is an optional command which allows a reader to transition a tag from the Open state to the Secured state or to remain in secured if the tag is already in the secured state.
- ***Authenticate*** command is an optional command that allows a reader to perform tag, reader, or mutual authentication. The generic nature of the Authenticate command allows it to support a variety of cryptographic suites. The CSI parameter (Cryptographic Suite Indicator) specified in an Authenticate command selects one cryptographic suite from among those supported by the tag.
- ***AuthComm*** command is an optional command that allows authenticated communications from Reader-to-Tag by encapsulating another command and typically also a MAC in the AuthComm's message field. The cryptographic suite indicated by the CSI in the Authenticate or Challenge that preceded the AuthComm specifies message and reply formatting.
- ***SecureComm*** command is an optional command that allows encrypted communications from Reader-to-Tag by encapsulating another, encrypted command in the SecureComm's message field. A tag may encrypt and/or include a MAC in its reply.

3.4 The EPC tag requirements

RFID tags usually have constrained capabilities in every aspect of computation, communication and storage due to the extremely low production cost. In this section, the RFID tag requirements are discussed.

3.4.1 Functional Requirements:

The primary functionality of the EPC Gen2 protocol is for fast and efficient tag identification across a range of operating environments. The tag microchip is responsible to perform the physical interaction between the reader and the tag, and to perform the logical operating procedures and commands defined by the standard. At least the mandatory commands defined by the standard have to be implemented.

3.4.2 Security Requirements:

The real-life applications necessarily require reasonable security and privacy levels with minimal possible cost. Authors in [84, 55] list some of the main risks, which require using cryptographically secure RFID tags, as follows: tag impersonation, information leakage, privacy concerns such as physical tracking of persons, and tag forgery.

The EPC G2 standard defines so called "cryptographic suites" that allows the system designer to add security services depending on the application needs. The following security services are chosen to be supported in the proposed design to provide general security functionalities:

- Random number generation
- Data encryption/decryption of the communication between the reader and the tag.
- Data encryption/decryption of the data stored in the memory
- Mutual authentication

3.4.3 Performance Requirements:

The standard specifies data rate range of 5-640 kbps for Tag-to-Reader Communication and range of 26.7-128 kbps for Reader-to-Tag communication. The chosen frequency for the digital baseband microchip part is 2 MHZ, using 2 MHz clock frequency allows reasonable throughput for passive tags [29, 92]. The throughput range using this frequency is between 1/400 to 1/3 bit per clock cycles. The maximum throughput is less than 1 bit

Table 3.2: T_1 timing and available clock cycles

Tari Time (μ Sec)	T_1 Time (μ Sec)	Available clock cycles
6.25	39.06	78
12.5	78.125	156
25	187.5	375

per clock cycles and this rate could be achieved using such extremely low power devices.

Tags have to answer the reader within specific response times. This times are usually very short, i.e., 15-250 μs for EPC Gen2 tags (nominal range). The responses time are defined in the standard and it depends on the reply type the received command, other parameters in the standard. There are three tag reply response time options defined by the standard: immediate, delayed, and in-process response time.

The reader begins a command by issuing a preamble that defines the length of the logic 0 and logic 1 symbols. The length of the logic 0 symbol is referred to as Tari, which is a fundamental timing parameter for communications.

The Tari value determines the amount of time the tag has to begin its response to the reader after the reader has completed the last symbol in its command to the tag. This time is referred to as T_1 time for an immediate tag reply response time. The following Table 3.2 shows the T_1 timing for the minimum Tari value of $6.25\mu s$, the maximum Tari value of $25\mu s$, and a commonly used Tari value of $12.5\mu s$. The clock frequency of the digital baseband operates on 2 MHz.

3.4.4 Area Requirements:

The area a chip depends on many factors, these factors include the required security level, intended market, cost of fabrication and size of production.

The RFID tag has to be small, extremely cheap chip as it will be attached to a physical object and is used to identify and store information about the object. A rule of thumb [84] sets that the hardware implementation for the cryptographic circuitry in the RFID tag must be under 2000 GEs, including the cryptographic primitives and supported memory banks.

3.4.5 Power Requirements:

RFID tags are passive devices which they receive energy from signal emitted by reader. A passive device should not consume more than 30-50 μWatts of energy [84]. The power usage for the digital baseband part should be less than $1-10\mu W$ average with peaks below $3\mu W$ and $30\mu W$ respectively [84]. Thus, at 2MHz, peaks should be below $1.5\mu W/MHz$ to $15\mu W/MHz$. Tags typically do not perform operations unrelated to communication during the communication phase due to the lack of power harvesting.

3.5 Related Work

The related work is divided into two sections: security for RFID tags, and hardware architectures for RFID tags. Many security protocols are proposed to bring cryptographic security to RFID tags. These protocols include using traditional cryptographic standards such as AES, or using lightweight cryptographic primitives. The hardware architectures for RFID tags can be implemented as follows: using specialised hardware circuits, using software implementations, and using co-processors. Detailed comparisons of our HSM to other research on adding cryptographic functions to EPC tags are in Sections 6.3 and 6.4.

3.5.1 Security for RFID tags

Many hardware cipher designs and security protocols have been proposed to bring cryptographic security to RFID tags. Many symmetric schemes are used to implement different security functions in RFID tags as Data Encryption Standard (DES)[33], Advanced Encryption Standard (AES)[22], Present[16], International Data Encryption Algorithm (IDEA)[60], etc. A low-area and low-power AES encryption engine is proposed in [18]. Reyhani-Masoleh et al. [79] proposed a single construction of AES combined S-box/inverse

S-box that is shared between the encryption and decryption data paths of the AES. Their design is smaller and faster than Canright's design[20] and is suitable for low-area AES designs. Yu et al. [93] designed some low-area and low-energy lightweight implementations for AES and PRESENT that are suitable for EPC RFID tags. Some authentication protocols using AES are proposed in [71, 68, 39, 34].

Public-key algorithms require large amount of computation, making it hard to be used for low-cost RFID tags. However some papers proposed public-key algorithms to be used in RFID systems. In [56, 6, 41, 58], RFID authentication based on elliptic curve cryptography is proposed. In [46], RFID security protocol based on RSA e-signature is applied in e-ticket. Arbit et al. [9] proposed two low-resource implementations of a 1,024-bit Rabin encryption variant in software and in hardware. Their implementation has a data-path area of 4,184 GEs, an encryption time of 180 ms and an average power consumption of 11 μW, these results fulfils the design requirements for passive RFID tags. Hinz *et al.* [51] showed their implementation of the RAMON cryptographic suite. This suite uses the Rabin-Montgomery (RAMON) public-key scheme to authenticate tags against the readers. In [78, 90, 13], the authors used elliptic Curve Cryptography (ECC) to implement security functions in RFID tags. The area of these public-key scheme designs are usually large. For example, the area of public key based secure identification protocol with different field sizes range from 8,582 GEs to 10,933 GEs[13], which is much more than areas of symmetric-key ciphers used in our HSM.

Adding security in such constrained devises as RFID tags are challenging. Many hardware designs are proposed to be suitable for the passive RFID tags that need optimized hardware implementations, good power management, and strong security capabilities as in [5, 91, 4, 94]. These lightweight ciphers are used to build different hardware models for our ASIP HSM. Ertl *et al.* [30] presented a design of the digital part of a security enhanced UHF RFID tag that conforms EPC Gen2 standard. The tag provides a mutual authentication functionality based on challenge-response protocol. In [61], the authors designed a system-on-chip UHF RFID tag IC for secure RF identification applications based on EPC Gen2 standard. They introduced some efficient power management techniques including a low voltage band-gap, a low drop-out regulation with a bias-boosted gain stage, and an adaptive DC limiter. In [86], the authors provides a battery-free platform for sensing and computation that is powered by UHF RFID reader signal conforming EPC Class-1 Gen-1 standard. They used a fully programmable 16-bit MSP430 micro-controller for the computations. In [69, 72] papers, novel secured lightweight mutual authentication protocols are proposed. Both protocols operate under the EPC Class-1 Gen-2 standard.

In order to prevent unauthorized access to the data on the tag or to clone tags by copying the unique identifier (EPCvalue) from one tag to another, a variety of lightweight security extensions for the EPC Gen2 standard have been proposed. Some lightweight security protocols and engines are proposed to reduce the complexity of computations and to make it more convenient for the passive RFID tags. In [24, 89] , the communication between the readers and tags are verified with the Cyclic Redundancy Check (CRC) function. Hummingbird algorithm was proposed in 2009, which is a special cryptgraphic algorithm suitable for RFID security [27]. In [63], the authors designed a digital baseband with AES cryptography engine, and the power consumption is optimized. In [80], an RFID Baseband Processor is designed and it can operate sensors. These protocols can be implemented in hardware or in software. They can be added into our HSM as extended hardware units.

3.5.2 Hardware architectures for RFID tags

There are different ways to implement RFID tag devices. The most conventional way is to use a specialised hardware circuits. Specialised circuits contain cryptographic function units and control unit circuits. The great advantage that this approach guarantees best silicon area, lowest power consumption, and best performance. However, there are two main problems: specialised hardware circuits provide poor or non-existent flexibility, and it needs large design efforts which increases the time-to-market compared to other solutions. Because of these two problems, many other architectures could be used to compensate these two drawbacks. The other two architectures that support flexibility for tag devices are to use software-based architecture (i.e. using general-purpose processors or embedded systems processors) or to use application-specific instruction-set processors. The following sections show some related works that have different architectures.

3.5.2.1 Software-based approach

Plos et al.'s RFID tag design [73] is one of the designs that is implemented using the software-based approach. The authors used a fully synthesizable 8-bit micro-controller that performs, in addition to the communication protocol, various cryptographic algorithms all in software. More discussions and evaluations for their design are discussed in Sections 6.3 and 6.4. Sample et al.'s RFID tag design [85] uses a fully programmable 16-bit MSP430 micro-controller for the computations but their design lacks security features.

The authors focus on building complete passive RFID platform called WISP to perform temperature sensing operations. Their software can be described on three levels: packet decoding and encoding, state and power management, and application layer protocol for encoding sensor data. The WISP platform was used in [21] to implement RC5 encryption in a passive RFID tag conforming the EPC UHF C1 Gen1 standard. They show that WISP has enough computational power to implement RC5 cryptosystem. However, their experimental platform exceeds the EPC UHF C1 Gen1 tags in terms of computing power and storage. The designs in this category are much slower and consume more energy compared to the other approaches. The main advantages of using the software-based approach is the high flexibility and the relatively short time to market.

3.5.2.2 Specialized hardware approach

Ertl's [30] and Fu's [39] designs are another RFID tag designs that use the specialized circuits approach. Ertl et al.'s main goal was to enhance security for passive RFID tags. The digital part of their security-enhanced tag including AES and Grain modules can be implemented with 12,000 GEs (without the non-volatile memory). Fu et al.'s RFID tag [39] performs both protocol handling and authenticating protocol using AES. The whole tag beseband design fits in 26,944 GEs. In general, the specialized circuits approach designs have the lowest area and power consumption and the highest performance for EPC tag designs, however it lacks flexibility and extensibility.

3.5.2.3 ASIP-based approach

Similar to our work, Groß et al. [42] use ASIP-based approach. Groß et al. implemented symmetric-key algorithms on a constrained 8-bit micro-controller. The authors analyzed the three block ciphers: Present, SEA, and XTEA. The area overhead for these ciphers is between 519 to 1,021 GEs (9.8% to 19.3% of the 8-bit micro-controller area). More discussions and evaluations for their design are discussed in more details in Sections 6.3 and 6.4. The ASIP-based approach allows for division of the functions between hardware and software. Even the hardware design for the processor is designed to work effectively with the desired application. This approach achieves balance between high flexibility and high security with low cost and high performance designs.

Chapter 4

System level design

The tag chip contains analog RF frontend, and digital baseband. The RF part is used to perform two-way communication with the reader and harvest energy from the reader's signal. The digital baseband is used to process all the commands and data. In our design, the digital circuit part are divided into two modules: the communication module (CM), and the hardware security module (HSM). This chapter discusses in details the functions and the responsibilities of both modules and how each module interacts with the other module. We designed and implemented the HSM module.

4.1 Top-level design

The digital circuit part in the RFID tag interprets the received demodulated data, performs the required actions (e.g. write data), and generates the appropriate responses. In our design, the digital circuit part are divided into two modules as shown in Figure 4.1: the communication module (CM), and the hardware security module (HSM). The CM is responsible for decoding the incoming commands, performing all non-security related operations, and encoding the replies from the second module (HSM) but the HSM performs all security related operations including read/write accesses to the memory.

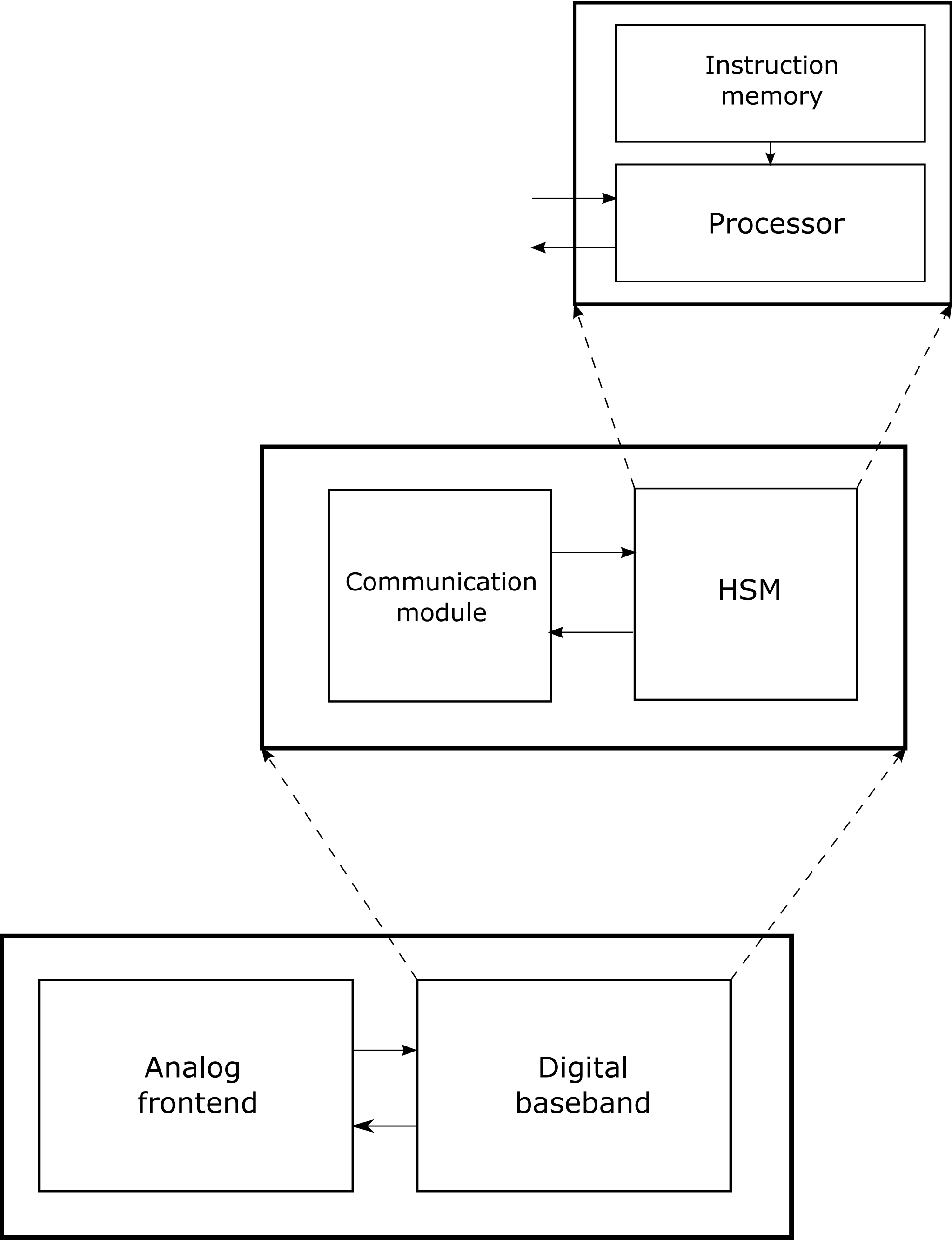


Figure 4.1: Tag device components

The division in CM and HSM has many benefits in terms of security. All security related functions and data are isolated in HSM module. The whole memory that stores all sensitive data and keys is accessed directly only by the HSM module. Furthermore, the division allows for more efficient implementations of hardware/software countermeasures against related implementation attacks like side channel attack, and gives more control in power consumption compared to that in stand-alone module architecture. The overhead hardware circuits or software codes to implement these countermeasures are limited only to the secure part. This saves area in hardware, power consumption, and even execution time. Side channel attack is one example of the related non-invasive attacks that aims at extracting sensitive data from a hardware structure by measuring physical characteristics like power, electromagnetic emission time delay. The division in two modules makes it harder for an adversary to gain sensitive data stored in the memory like secret cryptographic keys.

Our HSM module uses flexible ASIP-based architecture. The ASIP-based approach can achieve balance between rich functionality, low cost chips, and reasonable security requirements. The current passive RFID tag designs use the dedicated hardware circuits approach. This approach can achieve the lowest area and power consumption for RFID tag designs; however, it lacks flexibility. On the contrary, a system that uses the software-based approach is much slower and consumes more energy. However, the main advantages of this approach is the high flexibility and the relatively low time to market compared to other approaches.

4.2 The EPC tag digital baseband responsibilities

The digital part of the tag microchip is responsible for interpreting the received data then performing the required actions (e.g. writedata), and generating appropriate responses. In our design, the digital baseband is divided into hardware security module (HSM) and communication module.

The received input is interpreted and decoded using the communication module to identify the received command and its associated data, then the communication module sends a command to the HSM using interrupt vector connected between the communication module and the HSM. The HSM uses ASIP architecture so it acts as a processor which some interrupt service routine are written in the main memory, these interrupt service routines implement different actions required by the tag device. The required action

is performed using the HSM and the required response fields are forwarded to the communication module that combines or encodes them to generate the final response with the calculated CRC field as required for some commands.

An example of *Query* command sent from Reader to tag and tag's reply is shown in figure 4.2. The tag in figure 4.2 shows the digital part only. The Reader sends a *Query* command that initiates an inventory round. The tag picks a random number and sends it back to the Reader. The CM module decodes the received *Query* command. It processes all non-security related functions and sends an interrupt request (IRQ9) to the HSM. The HSM updates the program counter to a certain location according to the corresponding stored address to IRQ9 in the interrupt vector table. The start address of this interrupt service routine (ISR) is labelled in figure 4.2 as (Req_RN16). This ISR generates a 16-bit random number using custom instruction *prsg.run*. The generated random number is sent to the CM through output register r_1.

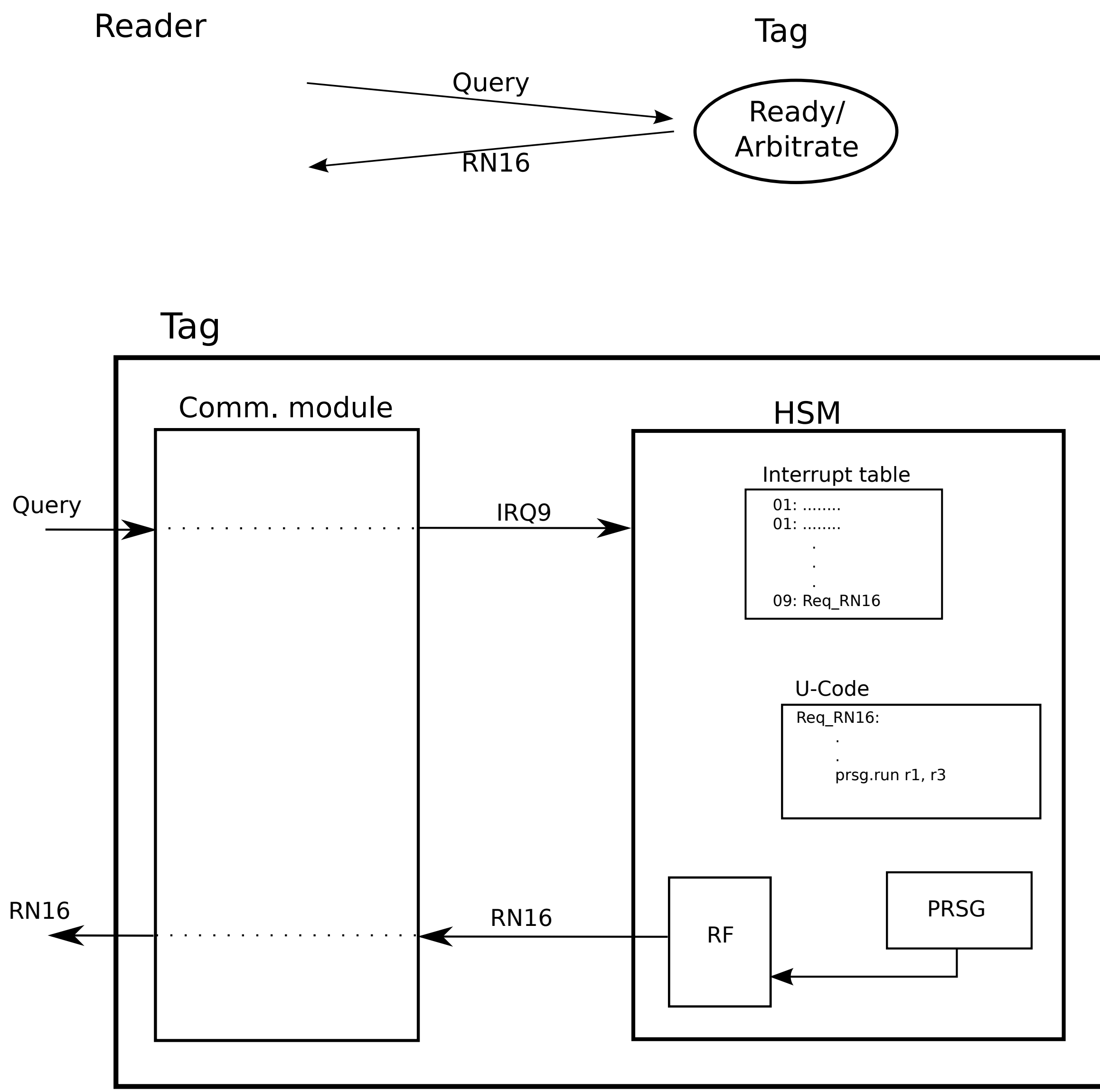


Figure 4.2: Tag digital baseband architecture

The responsibilities in the tag are divided between the communication module and the HSM. The following table 4.1 summarizes the responsibilities in each module.

Table 4.1: Responsibilities in the communication module and the HSM

Command	The communication module	The HSM
Shared tasks	- Decoding the received EPC commands - Encoding the final response - Processing the non-security related operations - Sending IRQs to the HSM	- Reading/writing the data parameters from/to the data memory - Sending the processed security related parameters to the CM
Select	- Asserting/Deasserting SL or inventoried flags - Forwarding Mask to HSM - Verifying CRC16	- Reading data from the specified memory - Comparing the read data to Mask
Query	- Verifying CRC5	- Loading/setting (Q) parameter - Picking a random value between 0 to 2^{Q-1} - Generating new random sequence bits RN16
QueryAdjust *QueryRep*		- Loading/adjusting (Q) parameter - Generating new random sequence bits RN16
Req_RN	- Verifying CRC16 - generating reply CRC16	- Verifying RN16 - Generating new random sequence bits RN16
Ack	- Forwarding RN16 to HSM	- Verifying RN16 - Reading Pc, EPC from the tag memory - Generating new Handle bits
Access	- Verifying CRC16 - Forwarding Handle to HSM - Forwarding access password to HSM - Generating reply CRC16	- Verifying Handle - Checking access password - Generating new Handle - Transitioning to the Secure/Open state
Read	- Verifying CRC16	

Table 4.1: Responsibilities in the communication module and the HSM

	- Forwarding the memory bank number to HSM - Forwarding the starting pointer to HSM - Forwarding the wordcount to HSM - Forwarding Handle to HSM - Generating reply CRC16 - Sending data to the analog front-end	- Verifying Handle - Reading data from the specified memory - Sending data - Generating new Handle
Write	- Verifying CRC16 - Forwarding the starting pointer to HSM - Forwarding the data to be written - Forwarding Handle to HSM - Generating reply CRC16	- Verifying Handle - Writing data to the specified memory - Generating new Handle
Kill	- Verifying CRC16 - Forwarding Handle to HSM - Forwarding Kill password to HSM - Generating reply CRC16	- Verifying Handle - Checking Kill password - Generating new Handle - Transitioning to the Killed state
Authenticate	- Verifying CRC16 - Forwarding CSI parameter to HSM	- Setting the CSI option - Performing required authentication action
AuthComm	- Verifying CRC16 - Decoding the encapsulated communication command	- Verifying the message and its MAC - Performing the encapsulated command action
SecureComm	- Verifying CRC16 - Forwarding the encrypted message	- Decrypting the encrypted message - performing the encapsulated command action

As shown in table 4.1, there are shared tasks that are performed with some or all of

the received EPC commands. The CM is responsible for decoding the received EPC commands and generating the corresponding interrupt requests to the HSM and forwarding the security related parameters to the HSM module. All non-security related operations are performed in the CM module. For example, some EPC commands require a reply that has a cyclic redundancy check (CRC). The CRC number generation is performed by the CM in our design.

The HSM is responsible for performing all security related operations as checking or verifying the received access/kill passwords or RN16 number. The generation of new RN16 or Handle is performed also in the HSM module. The HSM could include one or more crypto-modules that are used for performing cryptographic functions, these modules are accessed using the custom instructions that extend the supported instructions in the ASIP HSM. cryptographic functions could include random sequence generation, encryption/decryption, and hash function generator. The HSM also performs all read/write memory accesses and manage the accesses to the data memory depending on the security state.

4.3 The interaction between the HSM and the CM

The type of operations inside each module, the HSM and the CM, as well as the level of dependency between these operations in each module determine how the communication between the two modules is. Figure 4.3 shows an example of some operations performed by the HSM and some other operations performed by the CM for ACK EPC command. ACK command is used to transition to "Acknowledged" state. For valid ACK command, tag has to receive the last correct random number RN16. The green blocks in Figure 4.3 refer to the security related operations performed by the HSM. The other parts of the flowchart (blue blocks) are performed by the CM. The CM controls the command execution. Each green block means new interrupt request sent from the CM to the HSM. Any interaction with the memory is done by the HSM.

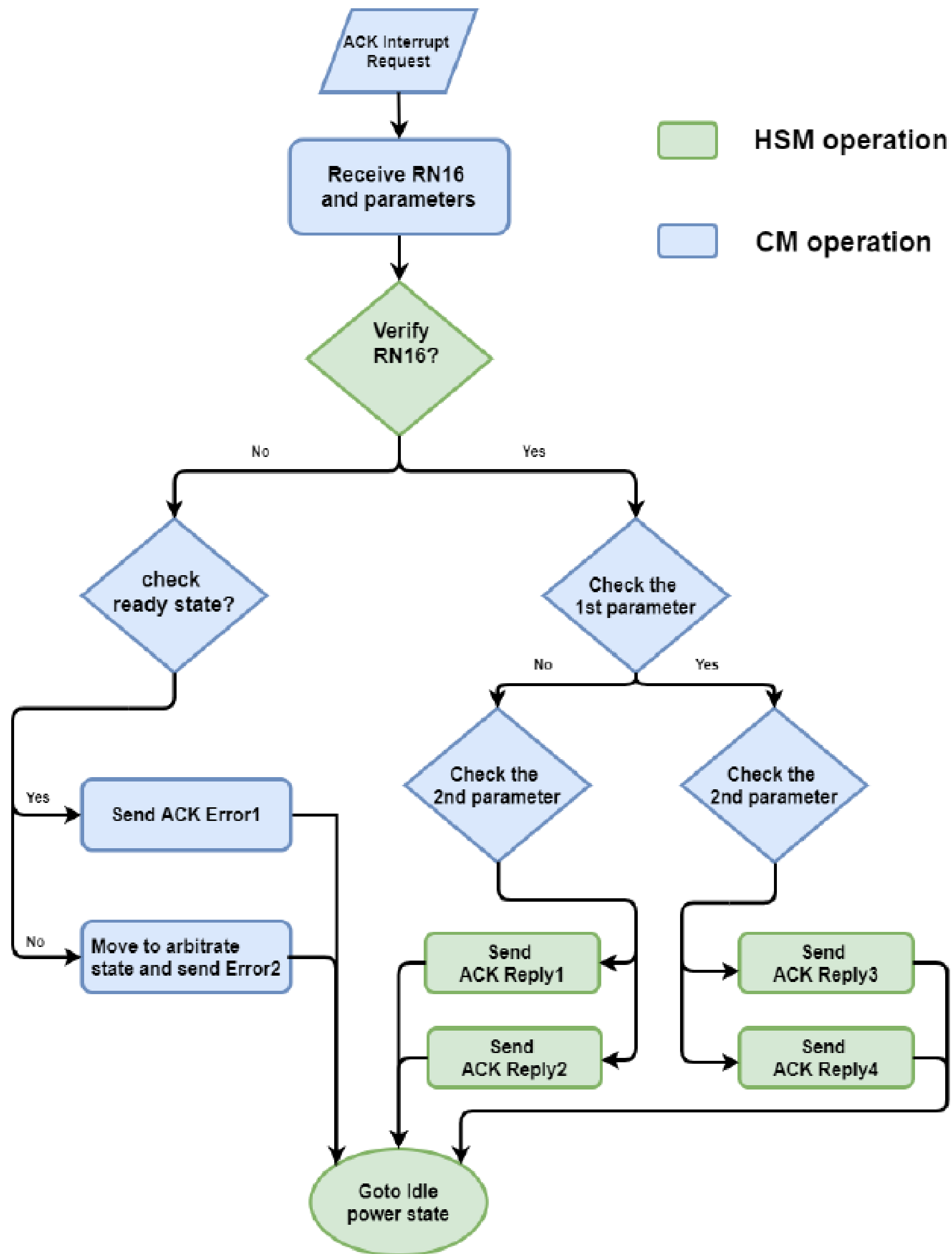


Figure 4.3: ACK command operations division between the CM and the HSM

The communication protocol between the two modules inside the tag upon receiving ACK command is shown in Figure 4.4. The CM starts decoding the two-bit ACK command, then it sends an interrupt request called "read_cmd" to read some data parameters from the data memory. Each data parameter requires sending new "read_cmd" IRQ from the CM to the HSM. The HSM starts loading the related parameters then sends them out to the CM. Then the CM sends the received RN16 (received from the Reader) to the HSM, it is noted that the maximum data rate for the Reader-to-Tag communication is 128 Kbit/sec which means throughput of 1/16 bit/clock cycle. The CM then sends new interrupt request to the HSM at clock cycle number 257 after sending all 16-bit RN16 to check whether this received RN16 matches the correct previous RN16 generated by the HSM itself. The HSM checks and replies with the check result, if it matches, it replies with correct RN16 to direct the CM to continue one of the execution paths. The CM checks the data parameters it has then ask the HSM to send one of the ACK reply sequences.

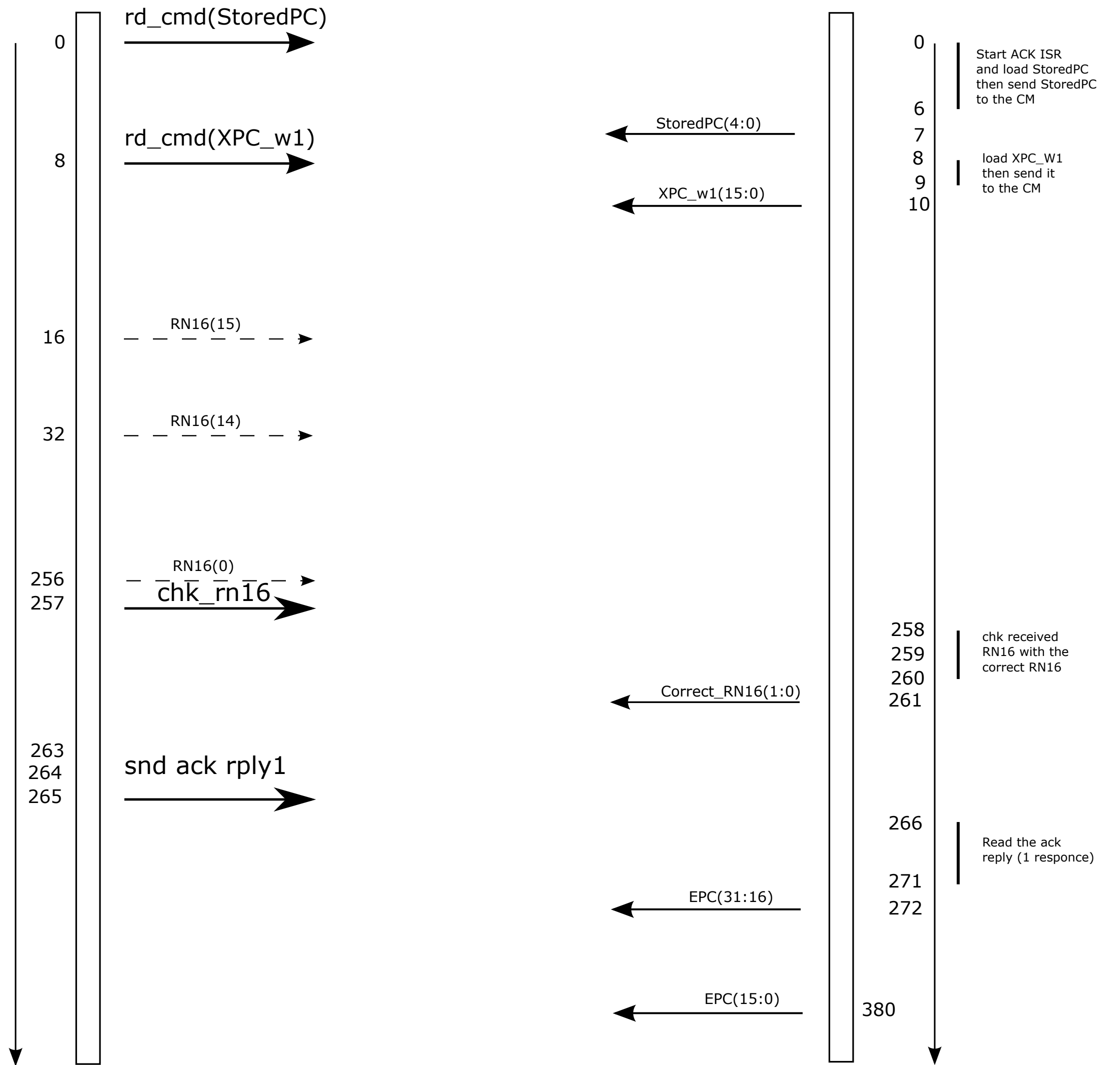


Figure 4.4: ACK command operations division between the CM and the HSM

Chapter 5

The HSM Architecture

This chapter discusses both the instruction-set architecture and the micro-architecture for our ASIP HSM. The instruction-set includes general instructions such as arithmetic instructions and special instructions such as custom instructions to access the extended hardware units. The chapter also discusses the micro-architecture features for our ASIP processor.

5.1 The overall architecture

Our ASIP processor uses reduced instruction set computing (RISC) architecture that has fixed instruction length of 14 bits and three instruction formats and has a single-cycle instruction implementation. The ASIP processor is able to perform general operations as well as special operations using the included hardware units. The added units can be accessed by custom instructions defined by the system designer at the design level phase (i.e. before fabrication), this adds more functionalities to the ASIP processor by only integrating the required units into the ASIP's datapath. The ASIP micro-architecture design is customized to minimize the hardware area complexity of the overall EPC tag device. The RISC architecture used in our ISA simplifies the instruction decoding hardware design.

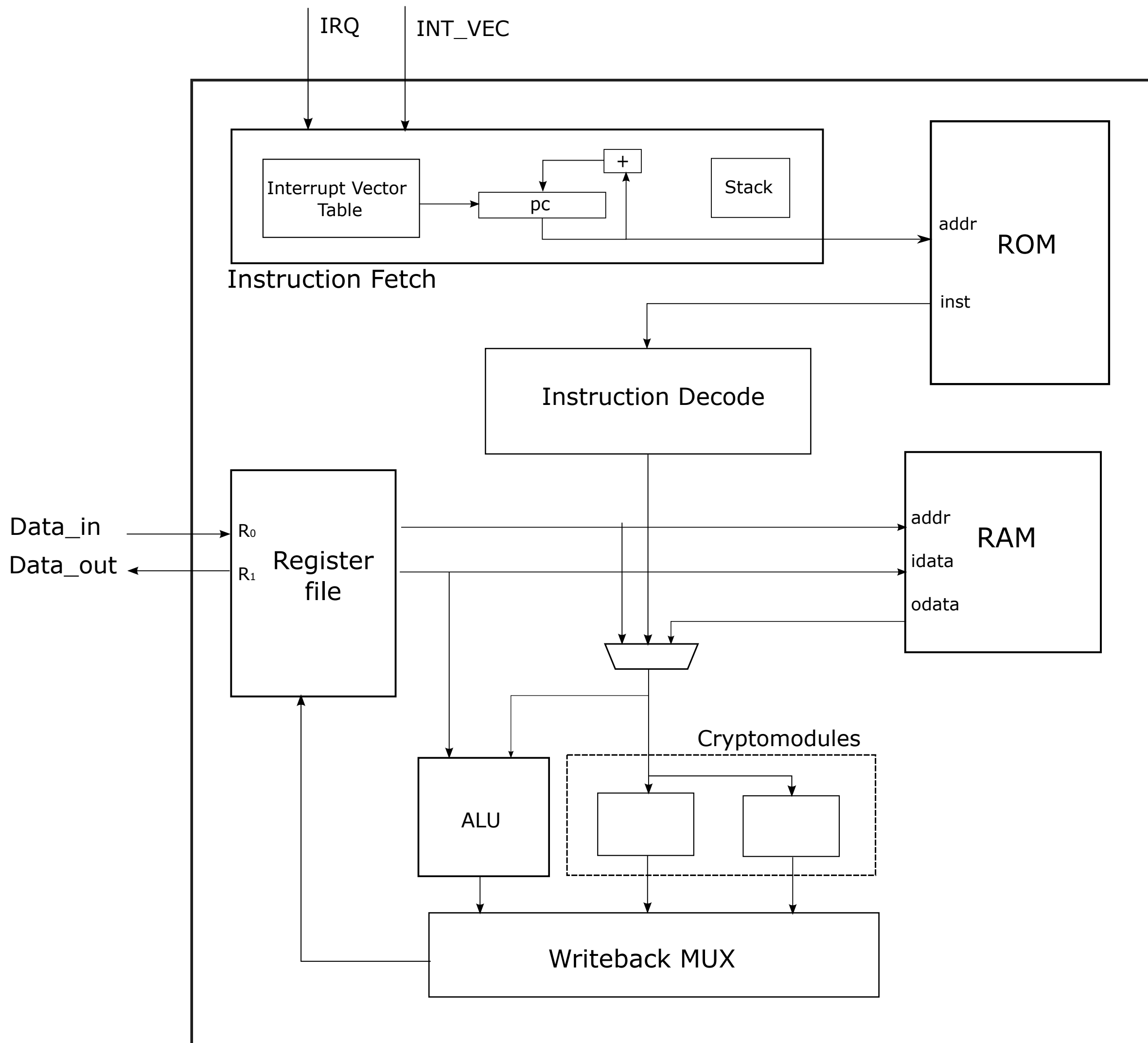


Figure 5.1: The microarchitecture datapath for the ASIP HSM

The code program of the EPC tag is divided into small code programs called interrupt service routines (ISRs). An ISR starts executing when the CM interrupts the ASIP HSM, the CM sends IRQ to start execution of some code program in the instruction memory. As shown in Figure 5.1, when the CM sends an interrupt request, the program counter (*pc*) is updated to point to the start address of the intended ISR. An instruction is fetched every clock cycle and the program counter is incremented as long as there is no branch control

instructions such as BEQ and JMP instructions.

Our ASIP processor architecture has some novel features, these features can be summarised as follows:

- Two registers in the register file are used for input/output operations, these registers allow for different serial/parallel loading modes.
- The accelerated mode support; the ASIP can repeat the execution of one instruction multiple times.
- Implicit operations can be run after writing some value to the status register or to the accelerated mode/output counter register.

5.2 Instruction-set architecture

The instruction set architecture (ISA) is a key design decision for the ASIP processor design. It is a primary goal for the passive tag designs to lower the overall implementation costs. We chose fixed-length instructions in our ISA, because it makes decoding circuit much simpler. Variable length instructions provide more compact code size but add more complexity to the decoding circuit and make organizing variable-length instructions in the ROM memory much harder. Our ASIP has two-operand architecture, this reduces the instruction length compared to that one in three-operand architecture. However, three-operand architecture gives more flexibility to use different source and destination registers. But it is more important to keep the instruction length as small as possible because this length affects the overall program ROM size. The chosen instruction length of 14-bits to cover all supported instructions. It is important to keep the instruction length as small as possible because this length affects the program ROM size. The instruction length depends on the number of instruction fields and their sizes, the number of operations, and the number of supported addressing modes. 14-bits are the least number that can be used for the instruction length to cover all supported instructions. Three instruction formats are used in our ASIP: Register-type (R-type), Immediate-1-type (I_1-type), Immediate-2-type (I_2-type).

From high level perspective, the ASIP HSM has to perform some general operations as well as some dedicated or special operations such as random number generation or

encryption/decryption. Hence, the ASIP instruction set requires the following types of instructions: move instructions, arithmetic instructions, jump/branch instructions, output instructions, custom instructions, and special control instructions. `LD`, `ST` instructions are used to move data between data memory and registers while `MOV` instruction is used to move data between registers. Arithmetic and logical instructions like `ADD` and `AND` instructions are needed for data manipulations. `LDL` is used to set the lowest significant 8 bits of a register by an immediate value. `SHL` and `SHR` are logical shift one-bit location instructions while `SHL8` and `SHR8` are shift eight-bit location instructions. `SHL8` and `SHR8` are used with output operations in R_1 register. `SHL8` is commonly used to set the most significant 8-bits of a register after `LDL` instruction. `BSR` is an instruction used to call a subroutine while `RTS` is used to return from a subroutine. `JMP`, `BEQ`, and `BNEQ` are jump/branch instructions for control flow and `PUT` is an explicit output instruction to output the contents in R1 to the output interface signal. `CUSTOM.XXX` as shown in Table 5.1 is just an example of the custom instructions that a system designer can add. The custom instruction(s) can have up to 16 options. The supported instructions are shown in Table 5.1.

5.3 Instruction Formats

There are three instruction formats supported in our ISA:

- Register format (R-type)
- First Immediate format (I1-type)
- Second Immediate format (I2-type)

5.3.1 Register Format (R-type)

There are two operand fields in this format, one field for the destination register and one of the source registers and the other operand field is for an optional another source register. The instruction of this format is divided into four fields: 2-bit opcode, 4-bit funct field (funct field is used to define an instruction or operation.), 4-bit destination register r_d (used also as a source register), 4-bit second source register r_s.

Table 5.1: The meaning and the encodings of the supported instructions

Instruction	Instruction format	Encoding	Meaning
`LD`	R-type	00.1110	Load from the data memory (RAM)
`ST`	R-type	00.1111	Store to the data memory (RAM)
`LDL`	I_2-type	10	Load immediate 8-bit value to a register
`MOV`	R-type	00.0110	Moving register content to a register

`ADD`
`SUB`
`AND`
`OR`
`XOR`
`NOT`
	R-type
	R-type
	R-type
	R-type
	R-type
	R-type
		00.0000
		00.0001
		00.0010
		00.0011
		00.0100
		00.0101
			Arithmetic and logic operations (add, subtract, bitwise and, bitwise inclusive OR, bitwise exclusive OR, bitwise negation)

`SHL`
`SHR`
`SHL8`
`SHR8`
	R-type
	R-type
	R-type
	R-type
		00.1001
		00.1000
		00.1011
		00.1010
			Arithmetic shift operations (logical left-shift, logical right-shift, logical left-shift 8 bit locations, logical right-shift 8 bit locations)

`BEQ`
`BNEQ`
`JMP`
	I_1-type
	I_1-type
	I_1-type
		11.0010
		11.0011
		11.0000
			Conditional branch (if equal, if not equal, if less than)
			Unconditional jump

`BSR`	I_1-type	11.1100	Branch to subroutine
`RTS`	I_1-type	11.1101	Return from subroutine
`PUT`	I_1-type	11.0110	Output data
`CUSTOM.XXX`	R-type	01.xxxx	Custom instruction

Table 5.2: Examples of custom instructions encodings and meanings

Instruction	Instruction format	Encoding	Meaning
`wg.run`	R-type	01.0100	WG-5 stream cipher encryption/decryption
`wg.prsg`	R-type	01.1000	Pseudo-random sequence generator
`wg.init`	R-type	01.0010	WG-5 stream cipher initialization
`wg.ldkiv`	R-type	01.0001	WG-5 stream cipher load key/IV
`simeck.run`	R-type	01.1100	Simeck block cipher encryption/decryption
`simeck.ldkey`	R-type	01.1101	Simeck block cipher load key
`simeck.lddata`	R-type	01.1110	Simeck block cipher load data
`ace.ldh`	R-type	01.0000	Ace load high 32 bits and read word0
`ace.rdw1`	R-type	01.0001	Ace read word1
`ace.rdw2`	R-type	01.0010	Ace read word2
`ace.rdw3`	R-type	01.0011	Ace read word3
`ace.ldl`	R-type	01.0100	Ace load low 32 bits
`ace.ldm`	R-type	01.0101	Ace load mode of operation and control bits
`ace.run`	R-type	01.0110	Ace run operation
`aes.ldkd`	R-type	01.0101	AES load key/data
`aes.ldm`	R-type	01.0110	AES load mode of operation
`aes.run`	R-type	01.0100	AES run operation
`aes.rd`	R-type	01.0111	AES read data

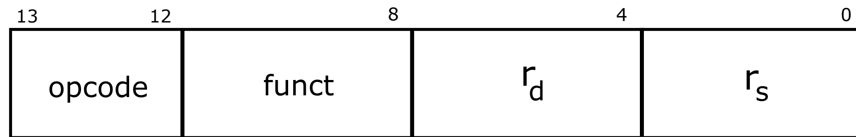


Figure 5.2: R-type instruction format

An example of instruction that uses this format is ADD instruction. The following code shows the assembly code for addition operation:

`add` r_d, r_s

Where r_d is the first source register and the destination register and r_s is the second source register.

The instructions that use R-type format are: ADD, SUB, AND, OR, XOR, NOT, MOV, LD, ST, SHL, SHR, SHL8, and SHR8 instructions and the custom instructions.

5.3.2 First Immediate format (I1-type)

In this format, there is one immediate value of 8 bits. The instruction of this format is divided into three fields: 2-bit Opcode, 4-bit funct field, 8-bit immediate (imm_8) as shown in Figure 5.3.

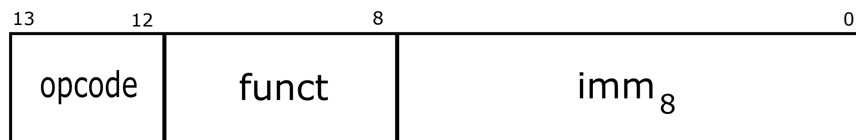


Figure 5.3: I1-type instruction format

The instructions that use I1-type format are BEQ, BNEQ, JMP, BSR, RTS, PUT instructions. The following code shows an example of an assembly code using BEQ instruction:

`beq` imm_8

Where imm_8 is 8-bit signed immediate, the imm_8 range is between -128 to 127. It will be added to the current pc to branch to the new added address in case of correct condition check or to the next instruction address in the other case.

5.3.3 Second Immediate format (I2-type)

In this format, there is one destination register and one immediate value. The instruction of this format is divided into three fields: 2-bit Opcode, 4-bit destination register r_d, 8-bit immediate (imm_8) as shown in Figure 5.4.

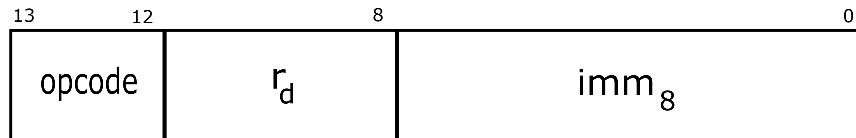


Figure 5.4: I2-type instruction format

The only instruction that uses I2-type format is LDL instruction.

The following code shows an example of an assembly code for loading an immediate value imm_8:

$$\texttt{ldl} \quad r_d, \quad imm_8$$

Where r_d is the destination register and, imm_8 is 8-bit data to be loaded into the lower byte of the destination register r_d. The load lower byte instruction (LDL) in the above assembly code has the meaning of

$$r_d \ = \ \{r_d[15:8], imm_8\}$$

5.4 Instruction format design decisions

Choosing the instruction formats is one of the important design decisions that affects on the whole program memory size and the overall performance.

5.4.1 Three-operand vs. two-operand ISA architectures

In two-operand ISA architecture, one of the two source operands (input operand) has the same register address as the destination operand (output operand). Registers in the register file are used mainly to store data temporarily. The data contents in these registers are used to feed other computational blocks either in the ALU or in the added cryptomodules. The following code shows an `ADD` instruction that uses two-operand architecture. R_5 is the first source operand that is used also for the destination. R_3 is the second source operand. The addition result will be written in R_5 register and will replace the old content in R_5 register (that is used for calculating the result).

$$\texttt{ADD}\ \ R_5,\ R_3; \quad R_5\ =\ R_5\ +\ R_3$$

The main benefit of using this two-operand architecture is reducing the number of fields in the instructions. Hence, reducing the program memory (ROM) size. In our implementation, using the two-operand ISA architecture saves about 22% of the overall instruction memory size of the three-operand ISA architecture. The main disadvantage of the two-operand architecture is losing some flexibility to keep the value of one source register without change. This may be a problem if there are many functions that require accesses to the same variables. The three-operand architecture will be much faster in this case because it keeps these variables in the register file then it will be used directly without new loads from the memory. This type of variables dependency is not the case in our system so two-operand architecture works well.

5.4.2 Stack-based vs. register-based ISA architectures

The advantage of a stack-based ISA is using short instruction words compared to that in a register-based ISA. There are not source and destination registers in a stack-based ISA. The awkward way of moving and manipulating data in a stack will add more clock cycles (i.e. more time) and will add more instructions (i.e. more instruction lines) to the program memory. There might not be a big saving in the program memory size. The most disadvantage is adding time overheads for all operations in the ASIP HSM. This time overheads make it hard to meet the latency requirements for EPC tags.

5.4.3 Variable length instructions vs. fixed length instructions

The main advantage of fixed length instructions with uniform formats is that decoding the instructions is obviously simpler. Instructions are aligned in the ROM memory as each instruction has one memory location and has same size as all instructions. A new instruction is fetched every clock cycle. The program counter is incremented every cycle (except when executing control flow instructions) independent of the instruction type. In variable length instructions, the advantage is larger code density. This may improve the code size of the program memory. Instruction length can be set according to frequency of use. For example, less encoding bits are given to most frequent instructions. However, fetching a single instruction can be more complex since the start of the instruction is likely to not be aligned. So buffering and shifting logics are needed. This adds complexity to the fetching and the decoding units. If the maximum instruction length is 16, the cost of these buffering and shifting logics would be a 14-bit register for instruction buffering (i.e. 50 GEs) and 16 15:1 multiplexers (i.e. 500 GEs). The granularity of the program counter for variable-length instructions would add more complexity to the decoding circuit. For example, it may be: 8-bits, 4-bits, or 2-bits. Because of the complexity of the fetch/decode circuits and the low probability of significant benefits, we chose to use a conventional fixed-length ISA.

5.5 The ASIP HSM features

The ASIP processor features can be summarized as follows:

- Instruction-set extensions.
- Implicit input/output operations.
- Explicit output instructions.
- Accelerated mode support.
- Interrupt driven control technique.
- Program code isolation by dividing it into interrupt service routines (ISRs).
- Memory banks hardware protection using passwords.
- Configurable physical addresses and memory spaces.

5.5.1 Interrupt driven control

Interrupt driven control technique is used to allow the CM to send a command to the ASIP HSM. The EPC Gen2 standard uses challenge response protocol, which means the reader sends an EPC command then the tag responds with a reply. In our architecture, the command is received and decoded by the CM then the CM sends an interrupt request to the ASIP HSM with or without associated data. Once the ASIP HSM is interrupted, it performs the required program and it responds with the appropriate data replies through data out signal and/or cmd signal. Wr_en, ack, out_en, and full signals control the communication between the CM and the HSM. The input/output interface signals are shown in Figure 5.5.

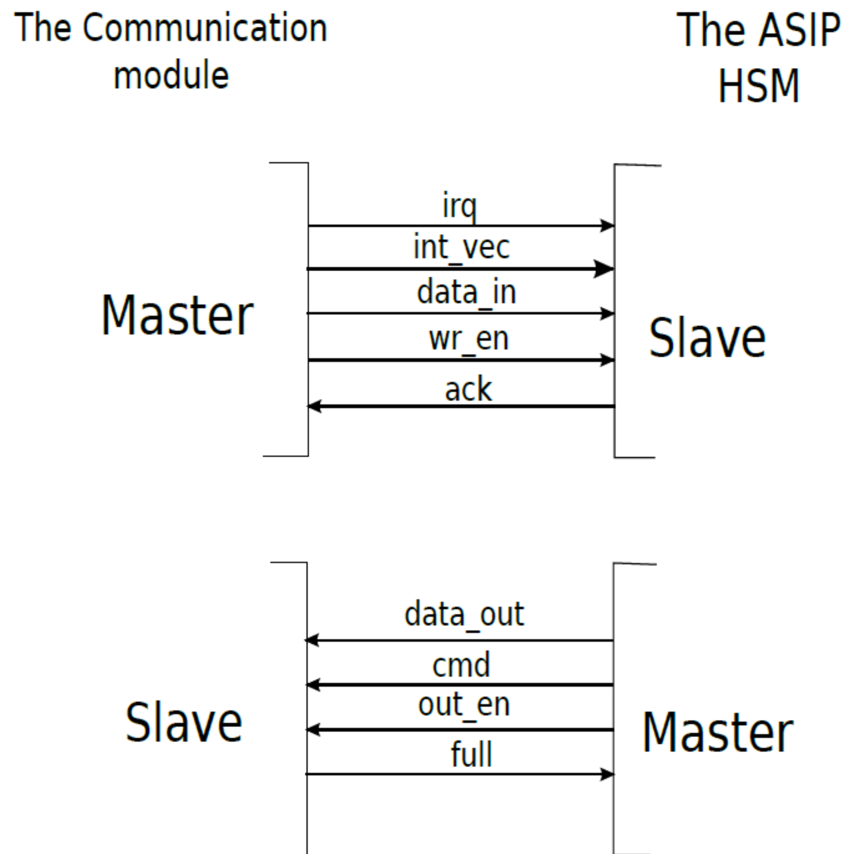


Figure 5.5: The input/output interface signals

The software program in the ASIP HSM is divided into sub-programs called interrupt service routines (ISRs). The following interrupt service routines have been implemented in the ASIP HSM:

- Verify the access password
- Verify the kill password
- Verify the 16-bit random number
- Generate 16-bit random number
- Generate arbitrary-length random number
- Save/restore state
- Read data word
- Read sequence of words
- Write data word
- Compare sequence of data
- Load keys / initialize the WG cipher
- Encrypt/decrypt using WG cipher
- Encrypt/decrypt using Simeck cipher
- Mutual authentication

5.5.2 Input/output operations

Registers R_0 and R_1 are used as input/output registers to receive or send data from/to the CM. The data in input has a path to R_0 input register. R_0 has special controls that allow three different ways for write operation:

1. Shift and rotate.
2. Shift logical left and write 1-bit to the least significant bit.

3. Parallel load (normal write operation as in all other registers).
4. Parallel load from the data input.

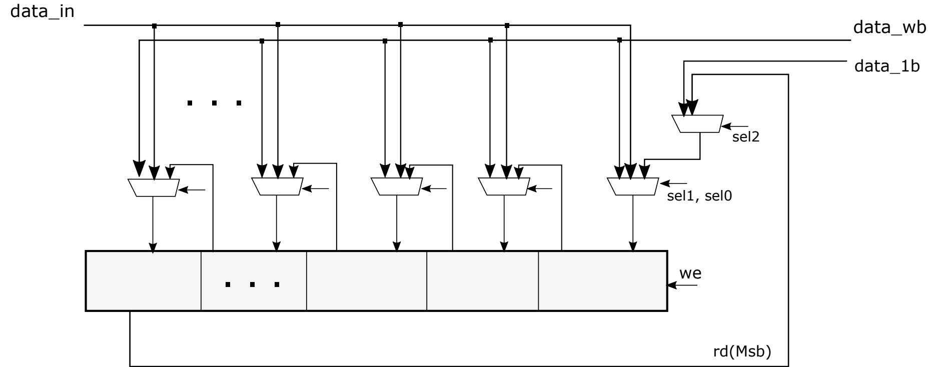


Figure 5.6: R_0 input register control logic

These different write options in R_0 register are adopted for the ASIP HSM operation needs. The first option (shift-left and rotate) allows the ASIP cryptographic module to perform an operation on a single bit (1-bit) from R_0 without using an explicit shift instruction in throughput of 1. This could be done by enabling what is called "accelerated stream mode" (this mode is described in more details in Section 5.5.4).

The second option is the serial input operation. In this option, 1-bit input data is written from the *data_in* input to the least significant bit $R_0[0]$ and shift-left. This option is needed to write a new input data stream from the CM module. A serial write operation from the data_in input is performed when the *sp_input* or the serial parallel input status bit is high and as long as *wr_en* is high and *ready* signal is high.

The CM module is able to send 1-bit data when the *ready* signal is high. The default value of *ready* output signal is high as long as R_0 has not been filled with 16-bit input data. This is monitored by *IP_CNTR* counter that has a default initial value of 16, the counter is decremented as long as there is an input operation and no read operation. *ready* signal goes low when *IP_CNTR* reaches the value of Zero. A Read operation from R_0 resets *IP_CNTR* counter by the initial value of 16.

The third option is the normal write operation when R_0 is the destination register of any instruction. The contents in R_0 could be accessed by a cryptomodule to perform 1-bit operation in "accelerated stream mode" as in the second option.

The fourth option allows for a parallel load for R_0 register from the parallel data input. This operation is done automatically when the *ready* signal is high and the *sp_input* status bit is low. The *sp_input* status bit is a bit that direct the load operation type in R_0 register (i.e. serial or parallel load operations). A parallel write operation from the *data_in* input is performed as long as *wr_en* is high and *ready* signal is high.

The read operation is done independently of the input or write operations. Any instruction that has R_0 as a source register could access the data content in R_0. In other words, there is no special input instruction to read input data. The sequence of the data input and the program execution that may read contents in R_0 is controlled by the CM.

R_1 output register has also special controls that allow other write operations. The output register R_1 is either accessed explicitly using the output instruction `PUT` as shown in Table 5.3, or implicitly by using R_1 as a destination register in an instruction. In both options, R_1 register could be loaded in serial or in parallel depending on the value of *sp_out* status bit in the status register. R_1 has two different ways for write operation:

1. Serial load and shift left.
2. Parallel load (normal write operation as in all other registers).

These different write options allow easy handling for the input or output data. For example, a stream cipher that takes its input from the serial data in input and sends the result to the serial data out output and it repeats 16 times. This operation can be done with enabling so called "accelerated mode" (described in Section 5.5.4) for this cryptographic instruction that performs this stream encryption operation. The different write operations to R_0 or R_1 registers allow flexible use of the special features in our ASIP processor. For example, our ASIP can decrypt a stream of data input bits and store the result in R_1 register in accelerated mode (i.e. throughput of one bit per cycle).

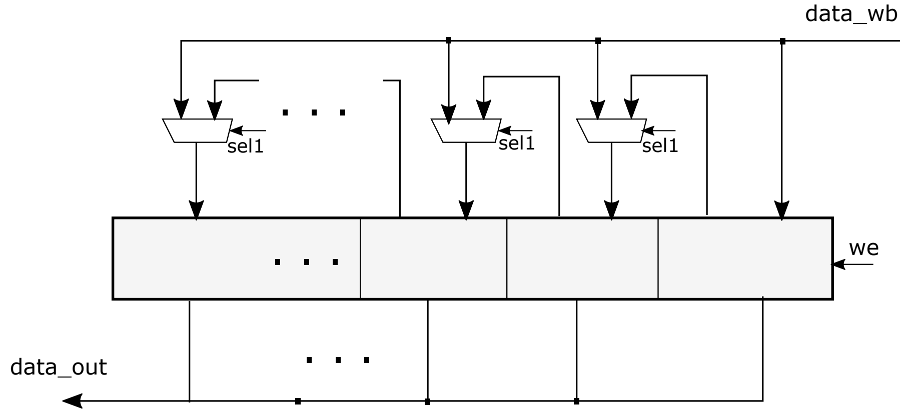


Figure 5.7: R_1 output register control logic

Table 5.3: Output instructions

Instruction	Assembly	Semantics
PUT	`ldl` r_1, imm_8 `PUT` #8	$r_1[7:0] = imm_8$ $OPCNTR \leftarrow 8$ While ($OPCNTR > 0$) {$R_1 \leftarrow R_1 << 1$ $OPCNTR \leftarrow OPCNTR$ - 1 }
Any R-type inst	`mov` r_1, r_s	$r_1 = r_s$

5.5.3 Instruction-set extension

The ASIP micro-architecture allows instruction extensions to the instruction-set architecture. The operations of these extended instruction are performed by the added hardware units. In our ASIP HSM, some hardware cryptomodules are added. Complex cryptographic functions could be performed in hardware in less number of clock cycles compared to that in software (i.e. without using cryptomodules). The ASIP allows the system programmer to access these included cryptomodules using simple instructions called custom

instructions. These instructions have reserved encodings for the extended operations. The added units are chosen to perform the required security functions for the EPC tag. The EPC G2 standard defines so called "cryptographic suites" that allows the system designer to add security services depending on the application needs.

The following security functions are chosen to be supported in this ASIP design:

- Random number generation.
- Data encryption/decryption of the communication between the reader and the tag.
- Data encryption/decryption of the data stored in the memory.

A cryptographic interface is used as a general interface for any added unit. This general interface makes it easy to include cryptomodules into the processors datapath. The general cryptographic interface, as shown in Figure 5.8, includes: two 16-bit input data, 16-bit output data, enable, input mode, output write enable.

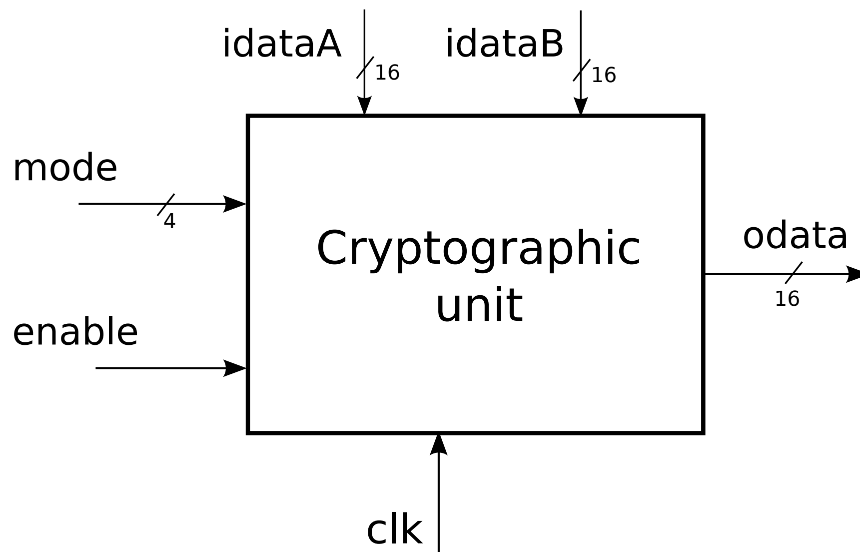


Figure 5.8: The input/output interface for a cryptomodule

An intermediate circuit called "cryptomodule wrapper" might be required to enable one operation in one module. In other words, this cryptomodule wrapper translates the

input/output signals of the general cryptomodule interface to the the input/output signals of the added cryptomodule. The design of these wrappers usually are simple. For example, the designed hardware wrapper for Simeck cipher consume less than 3% of the total area of Simeck cryptomodule as shown in Figure 5.9.

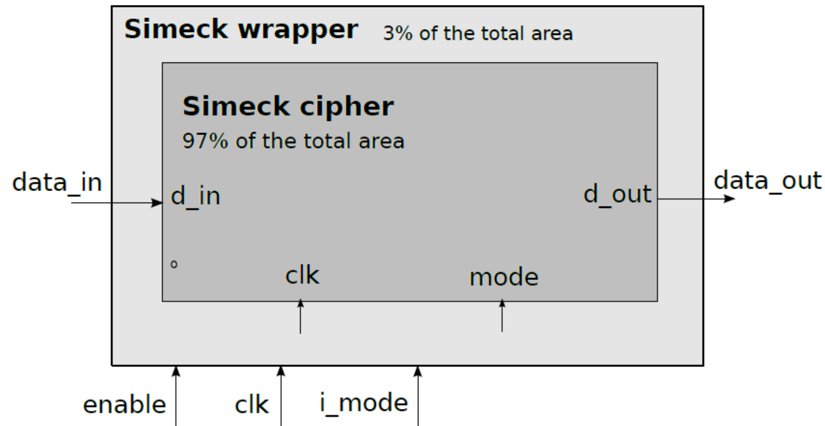


Figure 5.9: The included Simeck cipher in the ASIP processor

Our architecture allows up to 16 operations for the added cryptomodules. A *funct* field is part of R-type instruction format that is used to define an instruction or operation. The *funct* field is decoded from a custom instruction and forwarded to the mode input of all added cryptomodules. Each operation in any of the added cryptomodules is given an encoding value (i.e. a value from 0 to 15). A received *funct* field value will enable one operation in one of the added modules according to the assigned encoding values for these operations. The output from this module is sent to the write-back unit that may write it into the register file. The writeback unit uses the hardware output signal *wr_en* from a cryptomodule to write the *data_out* value into a destination register. The function of this *wr_en* output signal is determined by the system designer whom chooses the operations that write into a destination register.

This general interface allows easy integration for the added cryptomodules. The running operation has to be single-cycle operation so a full access is given to the program code to control the execution of the included modules and to control the sequence of the cryptographic operations. Table 5.4 shows assembly code examples as well as the semantics

for some instruction examples in this group.

The instruction set architecture design assumes that it does not know the operations in these cryptomodules. In other meaning, the ISA allows the system programmer to access these modules. It is the system programmer responsibility to use the appropriate instructions depending on the required functions/operations. In our ISA, we use a general instruction interface to use one input operand and one output operand that allows the processor to access the cryptomodules.

$$\texttt{CRYPTO.XXX}\ \ R_d,\ R_s; \quad R_d \leftarrow \ crypto_function(R_d{,}R_s)$$

In the following Table 5.4, it shows an example of different operations that may be used by a custom instructions called `CRYPTO`.

Table 5.4: Custom cryptographic instruction assembly code examples for different options

Instruction option	Assembly	Semantics
CRYPTO.INIT	`CRYPTO.INIT`	crypto_module.*mode* ← #init_crypto_module crypto_module.*i_valid* ← 1 crypto_module.*data_in* ← R_s (Data on *data_in* isn't used by the cryptomodule in this option)
CRYPTO.LOAD	`CRYPTO.LOAD` R_3	crypto_module.*mode* ← #load_crypto_module crypto_module.*data_in* ← R_3 crypto_module.*i_valid* ← 1
CRYPTO.RUN	`CRYPTO.RUN` R_4, R_3	crypto_module.*mode* ← #run_crypto_module crypto_module.*data_in* ← R_3 crypto_module.*i_valid* ← 1

Although the assembly code ignores the use of R_s and R_d in the first option (INIT) as shown in Table 5.4, R_s which is specified in the instruction field will be transferred to the cryptomodule data input interface. The enabled cryptomodule could use this data

on its *data_in* depending on the operation. The data on the *data_out* is written into the destination register R_d regardless there is a valid output data or not.

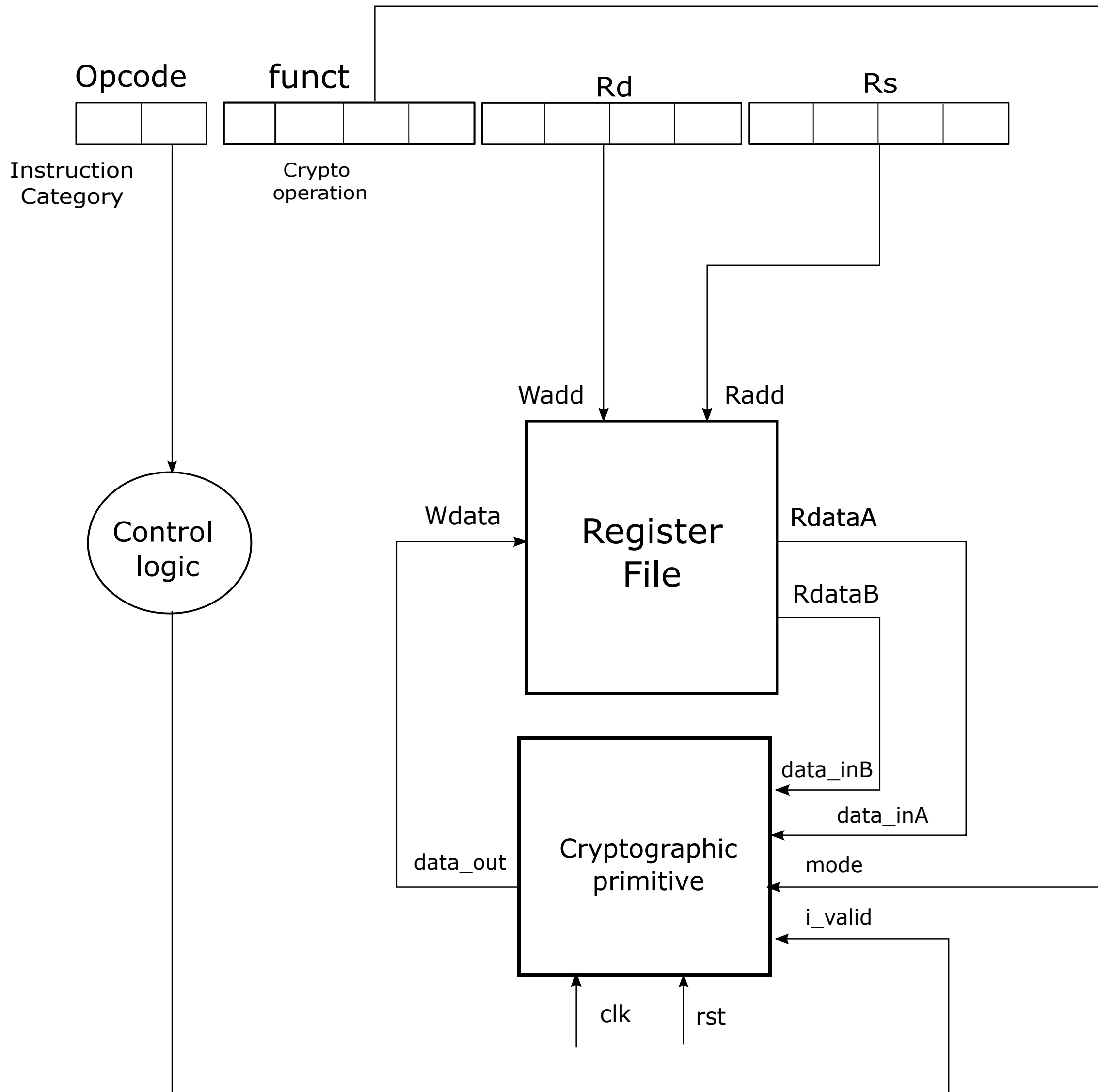


Figure 5.10: The cryptomodule interface connections

As shown in Figure 5.10, there is a field in the cryptographic instruction that determines the internal operation inside the cryptomodule. There may be one or more used

cryptomodules, the "Cryptographic operation" field is hardwired connected to the *mode* for the cryptomodule(s). The value of this field enables one operation in only one primitive that shares the same "instruction category" field value. So it works as module enable or as select operation. The important point here is that valid *mode* value as well as valid *valid_in* input signal are both required to enable the operation in one cryptomodule.

5.5.4 Accelerated mode support

Our ASIP processor supports so called "accelerated mode". In the "accelerated mode", the ASIP can repeat the execution of one instruction multiple times. This execution behaviour is common in the EPC tag operations. For example, sending stream of bits to the data output port or generating random number using a stream cipher. The throughput of 1 bit per cycle for EPC tags is suggested by [84] paper. There are many benefits of using the accelerated mode. The accelerated mode reduces the software program size by writing the accelerated instruction only once. Another way of writing repeated operations is using a loop.

In the accelerated mode, the same operations done in one iteration are performed in one clock cycle instead of three. This process can be done by setting R_{14} accelerated counter register using `LDL` instruction. The following instruction is performed multiple times as set by the `LDL` instruction. The PC is not incremented until the accelerated counter reaches zero. The accelerated mode works with other special features in the ASIP processor. For example, if the following instruction has destination of R_1 output register, the ASIP loads R_1 in serial or in parallel depending on the serial/parallel bit value in R_{15} status register. This instruction can be any of arithmetic or custom cryptographic instructions.

5.5.5 Instruction fetch unit and control stack

The instruction fetch unit is responsible for updating the program counter (PC) register. The PC is initialized from the interrupt vector table when an interrupt request is received. The PC increments as long as it doesn't reach the end of an ISR or there is no branch or call subroutine instruction. The program can call a function at any-time. Hence, the instruction fetch unit has a control stack. We have found that a stack size of 3 is sufficient for the ISRs that we have written. The stack size can be easily increased in the hardware

Table 5.5: Branch/control instructions

Instruction	Assembly	Semantics

BNEQ	`BNEQ` #12	if $Zero_flag == 0$ then
		pc ← pc + 12
		else
		pc ← pc +1

JMP	`JMP` $\# - 15$	pc ← pc - 15
BSR	`BSR` R_6	Stack[*sptr*] ← pc *sptr* -- pc ← r_6
RTS	`RTS`	*sptr* ++ pc ← Stack[*sptr*]

design process without affecting the ISA or other aspects of the hardware.

The branch/jump instructions update the PC to a new address. These instructions use pc-relative addressing mode to update the program counter. The new *pc* is calculated by the ALU by adding the current *pc* with the sign-extended immediate value in a branch control instruction. These instructions can be divided into two categories: decision making instructions, and call subroutine instructions. The instructions in the first category allow the ASIP processor to take decisions and branch to a certain execution path. This category includes `BEQ`, `BNEQ`, and `JMP` instructions. The second category is called "call subroutine" instructions category that allows the processor to jump directly to a subroutine and to return from a subroutine. There are two instructions in this category: `BSR`, and `RTS` instructions. Table B.3 shows assembly code examples as well as the semantics of some instructions in this group.

5.5.6 Special-purpose registers

The register file is divided into general-purpose registers and special-purpose registers. Registers R_2 to R_{13} are used as general-purpose registers. Registers R_0 and R_1 are used as input/output registers to receive or send data from/to the CM. R_{14} register is used as a counter for the accelerated mode or for the output operation. R_{15} register is status register.

R_{14} is used as a counter set by the programmer for the accelerated mode operations or for the output operations. It can be set using explicit output instruction `PUT`. It can be set using `LDL` instruction for the accelerated mode operation for the following instruction that is performed after `LDL` instruction. R_{15} status register includes status bits: tamper-detection bits, serial/parallel flag, secure bit, accelerated mode bit, internal/external write mode bit, memory bank bits, write enable bit, data input ready bit, and data output ready bit. Some status bits are accessed by the programmer and some other bits are written only by the ASIP processor control unit. Tamper-detection bits are set automatically by the ASIP processor if any of tampering attempts is detected. For example, counting mismatch memory passwords more than a certain limit or triple modular redundancy (TMR) fault is detected for the important registers in the register file as described in the following paragraph. The different tamper-detections circuits and the action responses (i.e. tamper resistance levels) are left as future work.

Some special registers are protected against fault injection attack by using TMR fault tolerant technique. This is implemented by repeating the hardware resources and the operations for these registers two more times. The TMR hardware control circuit masks faults using voting outcomes for the three repeated register units. This technique is used with the input register R_0 and the output register R_1 and the status register R_{15}. Using this technique for all registers is expensive as it adds two register units beside the voting hardware logics to the protected register unit. Hence, the most sensitive and important registers are chosen to be protected against fault injection attack.

5.5.7 Memory management unit (MMU)

The EPC Gen2 standard defines four distinct memory banks for data organization, which are known as tag memory. The four memory banks are: Reserved, EPC, TID, and USER. The Reserved memory contains the 32-bit kill and access passwords. EPC memory contains EPC code number. The EPC code number identifies the object to which the tag is attached. TID memory contains an 8-bit ISO/IEC 15963 allocation class identifier. It also contains some information for a reader to identify the custom commands and the optional features that a tag supports. USER memory is optional memory that may be partitioned into one or more files. The system designer defines the data stored in this memory.

The memory management unit (MMU) is responsible for mapping or translating logical addresses sent by the reader to physical addresses for the actual data locations in the RAM

memory in the tag. The tag and the reader use the logical addresses, which conform to the standard in their communications to access the data in the tag memory. The physical memory organization is set by the system designer whom is able to configure the starting addresses of these memory banks and the memory banks sizes using a dedicated small memory. This adds more flexibility for memory organization even after fabrication.

The MMU also provides a memory protection by matching the received internal passwords sent by the control unit and by controlling read/write accesses in different conditions. Each of the memory banks has its own stored password, which has to be matched with the received password from the control unit. Each memory bank has its own access rights. We divided the memory accesses into three categories: internal read/write accesses, external secure read/write accesses, and external open read/write accesses. Internal read/write accesses are used for the operations executed inside the HSM and do not include input/output operations. While external read/write accesses are the read/write operations that involve input/output operations. The access may be granted or denied depending on the access type (i.e. read or write), the access condition (i.e. internal or external), the security state bit, and the memory bank.

The permission bits for the different memory access type/condition are configurable using a small dedicated memory. Table 5.6 shows an example of the permission bits structure for the memory banks. For example, passwords in the Reserved memory bank can not be read externally (i.e. sent to the reader) regardless of the security state bit value but it may be read to execute password check operation, which is an internal operation in the ASIP HSM. In our implementation, The USER memory bank are divided into three sections in our implementation. Each section has its own permission bits. As shown in the example, UsrHigh memory section has more read/write constraints compared to that in UsrMED and UsrLOW memory sections. The starting physical addresses for the memory banks/sections are also configured by the system designer and stored in the same dedicated memory used for the permission bits.

Table 5.6: Permission bits structure example

Access type		Pwd	EPC	TID	User High	User Medium	User Low
Internal access	rd	✓	✓	✓	✓	✓	✓
	wr	×	×	×	✓	✓	✓
Secure external access	rd	×	✓	✓	✓	✓	✓
	wr	×	×	×	✓	✓	✓
Open external access	rd	×	✓	✓	×	✓	✓
	wr	×	×	×	×	×	✓

Chapter 6

Results and Evaluations

This chapter shows and evaluates the implementation results for our work. First, it discusses the synthesis results for our work and compares these results with the related other works that use different design approaches. Then, a detailed comparison for the architectures of some works that use programmable approach is explained. Finally, a detailed comparisons for the performance, and the optimality score results for different cryptographic algorithms implementations on these programmable works is discussed.

6.1 HSM architecture and results

Our ASIP HSM can be implemented with 4,662 gate equivalent units (GEs) for 65 nm CMOS process technology. This result excludes the memories (i.e. the data memory, and the instruction memory) and the added cryptomodules. Table 6.1 shows the detailed area results for the ASIP components. The register file has the largest portion in the area utilization between the ASIP components. The register file consumes 2,584 GEs (about 55.4% of the overall area).

Another implementation, using 11 16-bit registers, can be implemented with 3,860 GEs. This implementation result shows that changing only the register file size, from 16 to 11 registers, reduces the overall area by 802 GEs (17.2% reduction of the overall area of the 16x16 implementation). Register file size is considered as flexible parameter that is determined by a system designer. It depends on maximum number of used registers in written

subroutines. Our implementation can support up to 16 registers. These synthesis results are done with Synopsis Design Compiler using Synopsis CMOS65nm technology library.

Table 6.1: The chip area results for the ASIP design components

Component	Area [GEs]	Percentage
Control unit and memory controller	1,488	31.9%
ALU	590	12.7%
Register file (16x16)	2,584	55.4%
Total area	4,662	100.0%

The ALU supports 10 arithmetic 16-bit operations. The control unit contains instruction fetch unit, instruction decode unit, write-back unit, and memory management unit. The instruction fetch unit can update the program counter using different ways. Two small memories are used by the instruction fetch unit: Interrupt vector table, and control stack (3 registers). The interrupt vector table is customized to support only the number of implemented interrupt service routines (ISRs), 11 ISRs are used in this design. The instruction decode unit can decode the supported 20 instructions beside extended instructions using three instruction formats. The register file can have up to 16 16-bit registers. 4 registers are combined with some logic circuits to perform special operations (i.e. input/output operations, accelerated mode, and updating the status bits).

The overall implementation for the ASIP HSM was verified on hardware and software levels by running 11 security-related ISRs in the passive RFID EPC tag. The software program was written in assembly language for the ASIP HSM. A custom assembler tool has been implemented to transform code from assembly language to machine code. The overall software code for the security-related functions fits in 448 Bytes in the ROM memory. 11 ISRs have been implemented with average 22 lines of code for each ISR.

To demonstrate the flexibility of our apprach, we used cryptographic modules implemented by others as standane units. AES was implemented by Yu et al. [94, 93], Simeck

was implemented by Yang et al. [91], WG-5 was implemented by Aagaard et al. [5], and ACE was implemented by Aagaard et al. [4].

6.2 Other work results

The related work designs can be categorized into three groups: 1) a software-based approach, 2) a specialized hardware approach, and 3) an ASIP-based approach. In the software based approach, the communication protocol as well as the cryptographic algorithms are implemented all in software using general purpose processors. The second group of designs use specialized hardware circuits to perform tag functions including communication protocol and cryptographic algorithms. The ASIP-based approach divides tag functions between software and hardware. Table 6.2 shows some related work synthesis results for the three groups and main design features in each work.

Plos's[73] and Sample's[85] designs are RFID tag designs that use software-based approach. The authors in [73] used a fully synthesizable 8-bit micro-controller that performs, in addition to the communication protocol, various cryptographic algorithms all in software. They analysed the block ciphers AES, SEA, Present and XTEA as well as the stream cipher Trivium. Their micro-controller can be implemented within less than 5,600 GEs excluding the ROM. The ROM contains the program of up to 4,096 instructions (i.e. 8,182 bytes) and is realized as look-up table in hardware. Sample et al.'s RFID tag design[85] uses a fully programmable 16- bit MSP430 micro-controller for the computations but their design lacks security features. The authors focus on building complete passive RFID platform called WISP to perform temperature sensing operations. Their design provides over 8 kBytes of flash memory, 256 Bytes of RAM.

Ertl[30], Fu[39], Ricci[80], and Man[63] designs are RFID tag designs that use specialized circuits approach. Ertl et al.'s[30] main goal was to enhance security for passive RFID tags. The digital part of their security-enhanced tag including AES and Grain modules can be implemented with 12,000 GEs (without the non-volatile memory). The tag provides mutual authentication functionality based on a challenge-response protocol and the Advanced Encryption Standard (AES). The stream cipher Grain is used for generating cryptographically secure random numbers during the authentication procedure. The area of AES and Grain modules together are 6,150 GEs in [30].

Table 6.2: The related work implementation results

Reference	this work	Groß[42]	Plos[73]	Sample[85]	Ertl[30]	Fu[39]	Ricci[80]	Man[63]
design method	ASIP-based		software-based		specialized hardware			
operation	security	protocol & security	protocol & security	protocol	protocol and security			
technology (nm)	65	130	350	NA	180	180	180	180
area (mm^2)	0.038	NA	NA	MSP430	NA	0.149	0.205	0.446
area (GEs)	$4{,}662^B$	5,300	$5{,}600^B$	NA	12,000	26,944	NA	NA
Security circuitry (GEs)	Simeck 774 WG_5 1,258 AES 2,205 ACE 4,940	SEA 519 XTEA 608 Present 1,021	software	software	6,150	4,952	0.106 mm^2	NA
memory size	448B ROM 128B RAM	Protocol handling 2,142B ROM + Cryptographic algorithm code	≤8,192B ROM	8KB Flash 256B RAM	NA	NA	NA	136 bits
cryptographic algorithm	WG_5-80 Simeck-64 AES-128 ACE	Present-80 SEA-96 XTEA-128	AES-128 Present-80 SEA-96 XTEA-128 Trivium-80	NA	AES-128 Grain-80	AES-128	AES-128	AES-128

[A] NA: Not available
[B] Without hardware extensions and without memories.

Fu et al.'s RFID tag [39] performs both protocol handling and authenticating protocol using AES. The whole tag beseband design fits in 26,944 GEs. The circuit area of AES module is 4952 GEs. In [80], an RFID Baseband is designed. The authors integrated AES primitive aimed at secure data transmission. The overall area of ISO 18000-6C RFID baseband is 380 μm$\times$ 540μm, whereas AES engine requires 380μm$\times$ 280 μm (i.e. about 6,000 GEs). In [63], the authors designed a digital baseband with AES cryptography engine, and various low power design techniques are proposed to reduce the power consumption of the baseband of the passive tag. The area of their baseband system is $0.446mm^2$.

Groß et al.'s [42] uses ASIP-based approach. Groß et al. implemented symmetric-key algorithms on an 8-bit micro-controller. This micro-controller is the same one used in [73]. The authors analysed the block ciphers SEA, Present and XTEA as well as the stream cipher Trivium. They implemented software implementations without using hardware extensions (i.e. additional hardware units) and some other implementations using hardware executions. The micro-controller for a 130nm CMOS process technology has area of about 5,300 GEs without program ROM. For a 130nm CMOS technology, encryption and decryption operation of Present-80 can be implemented with overhead costs of 1,000 GEs, SEA-96 and XTEA with around 600 GEs. After Groß's[42] and Plos's work[73], there has not been new work that has focussed on programmable architectures to add ciphers to EPC tags. Most of the work has been on either new ciphers [59], [49] or protocols for authentication and other security services such as in [10], and [68].

6.3 Comparison between our ASIP and Plos's and Groß's processor architectures

This section presents a detailed comparison of our HSM to the work that is most similar to ours: Plos's software based approach and Groß's ASIP based approach. The 8-bit micro-controller used in Plos's, and Groß's designs is based on a reduced instruction-set computer (RISC) architecture with separate program and data memories. The instruction memory has a 16-bit word size while the data memory has an 8-bit word size. The program memory contains the program of up to 4096 instructions. Our customized processor also uses a RISC architecture with separate program and data memory. We have different word sizes for the memories. Our instruction width is 14-bit while the data memory has a 16-bit word size. The maximum instruction memory size based on our ISA is 2^{16} instructions. However, our data and instruction memories are scalable so their sizes could be adjusted

as needed.

The register file hierarchy is similar in our design and Plos's design. The register file in both designs contains a set of general-purpose registers and special-purpose registers. Size of the register file is flexible. Plos's processor has up to 61 general purpose registers and 3 special-purpose registers: an accumulator register (ACC) for advanced data manipulation, a status register (STATUS) that gives information about the status of the ALU, and a program-counter register (PCH) for addressing the higher 4 bits of the program counter. Some general purpose registers are used for advanced data manipulation (6 AMBA bus registers, IO register for direct signals). Our processor has up to 16 registers. Up to 12 of them are general purpose registers and 4 special-purpose registers. Special purpose registers contain two registers for input/output operations, one register used as a counter for accelerated/output operations, and the STATUS register.

Register file has the largest portion in the area utilization for our ASIP design and for the Plos's processor. As shown in Table 6.3 and in Figures 6.1 and 6.2, register file consumes 4,582 GEs in Plos's design; 81.7% of the total area, while it consumes 2,584 GEs; 55.4% of the total area in our design. The large number of registers used in Plos's design explains these results; they need all of these registers as their implementation for cryptographic algorithms are all done in software.

Table 6.3: The chip area results

	Our ASIP design		Plos's design	
Component	Area [GEs]	Percentage	Area [GEs]	Percentage
Control unit and memory controller	1,488	31.9%	747	13.4%
ALU	590	12.7%	265	4.7%
Register file (16x16)	2,584	55.4%	4,582	81.9%
Total area	4,662	100.0%	5,594	100.0%

Another noted difference is the control unit area in both designs. Our ASIP design consumes 1,488 GEs comparing to 747 GEs for Plos's design. The main reason is that

our design uses some hardware countermeasures to resist against relative implementation attacks. For example, our design has memory management unit that supports some access control mechanisms to secure accesses to the memories. Plos's does not support these mechanisms and the main work for security is performing cryptographic functions either in software or in both (software, and hardware).

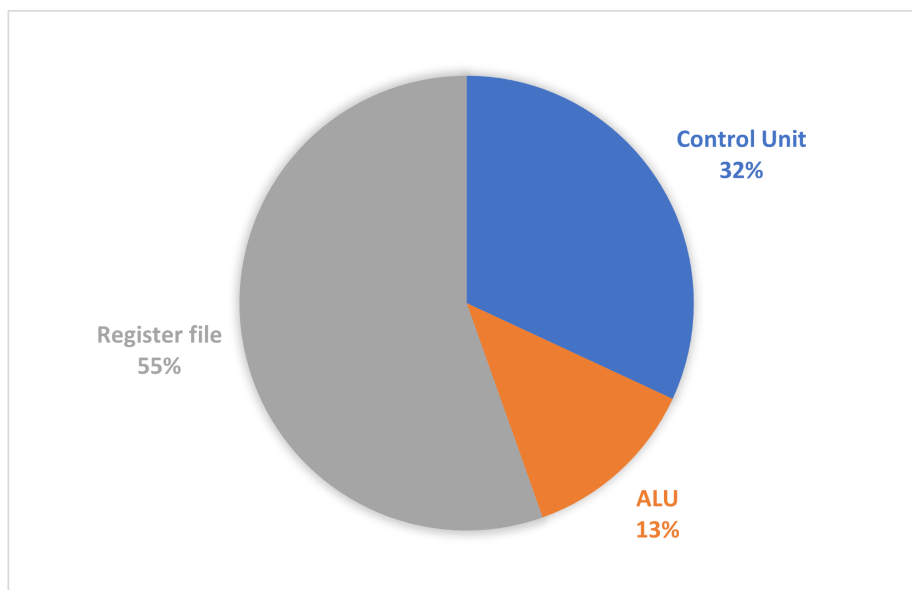


Figure 6.1: Our ASIP processor area components

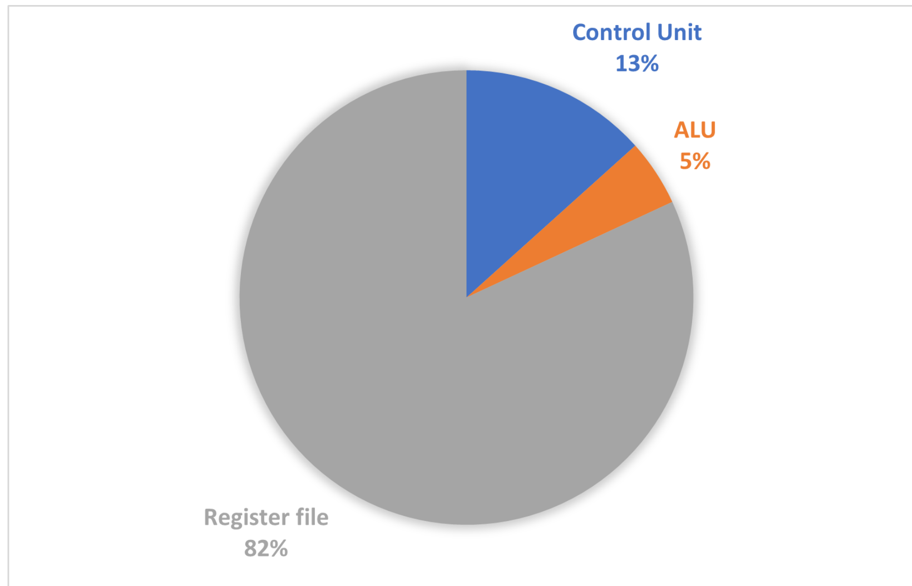


Figure 6.2: Plos's processor area components

Our design usually has smaller program size compared to Plos's or Groß's designs. This is clearly understood when we talk about the Plos's design as it uses only software approach (i.e. all functions are implemented in software) but what about Groß's design? In Groß's design, the hardware extensions they use are mainly to support a sub-operation of a cryptographic operation. For example, the authors use Sbox and Perm hardware extensions and use these modules to build Present-80 block cipher. In contrast, our design uses a complete cryptographic module and the ASIP is responsible to operate cycle-by-cycle for the added cryptographic module. Usually this way reduces the program size especially when using loops or accelerated mode operation. However, it may cost more hardware area compared to smaller modules that perform sub-operations only. This kind of added hardware extensions affects the number of temporary variables used by the processor. We have fewer registers compared to Plos's and Groß's designs.

Another noted difference between our design and the Groß's design: Groß's design uses a custom interface for each extended hardware they added into their design. For example, 8-bit data input/output interfaces are used in s-box hardware unit for Present-80 block cipher and 8x8-bit data input/output interfaces are used in permutation hardware unit Present-80 block cipher. Each extended module is accessed by a separate instruction. In our design, a general interface is used for all included hardware modules. This general

interface allows more flexibility to add hardware modules in such easy manner. The only needed effort to add a hardware module to our ASIP is to design a hardware wrapper that enables the module's operations when it is called by a custom instruction. More details about hardware extensions are discussed in Section 5.5.3.

The flexibility in our design allows for adding different cryptographic modules depending on system's need. Figure 6.3 shows the total areas for different HSM implementations. The design variable here is the added cryptographic modules. The overhead is varying from 13.3%, in case of adding Simeck module only, to 106.6%, in case of adding both ACE and WG-5 modules.

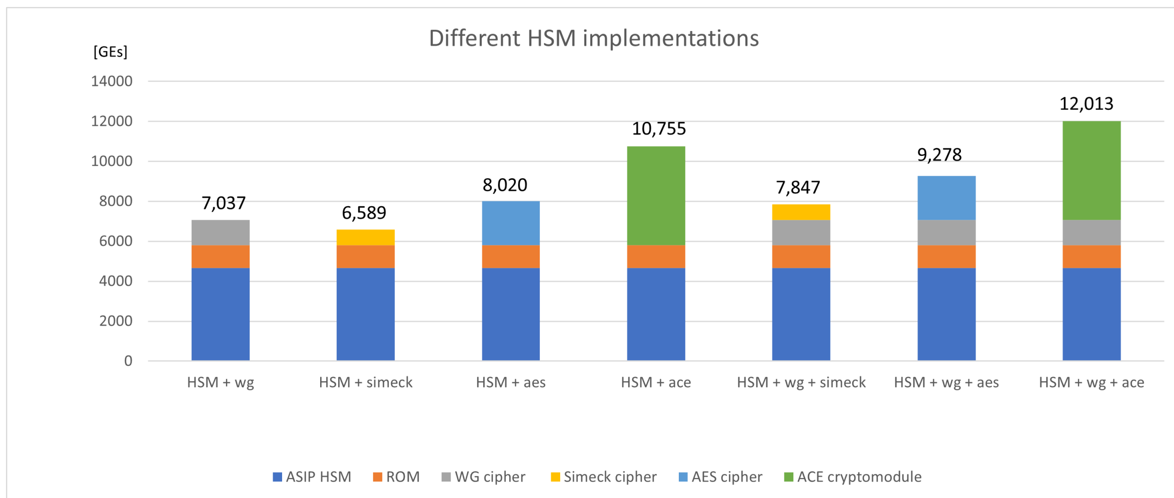


Figure 6.3: HSM implementations using different cryptomodules

Plos's micro-controller supports 36 instructions. All instructions are executed within a single clock cycle, except control-flow operations which require two clock cycles. Instructions are executed within a two-stage pipeline that consists of a fetch and a decode/execute step. In our design we support 20 instructions with up to 16 extended operations using the added hardware extension modules. Our design uses a single clock cycle for all instructions including the extended instructions.

The instruction operations categories are similar in our design and in Plos's microcontroller (i.e. logical, arithmetic, control flow, cryptographic instruction). However, there are

some differences. In Plos's micro-controller, the authors implemented various instruction options as separate instructions with distinct encoding. For example, some instructions have register to register operations. Some other instructions handle data coming from a constant or a register with data in the accumulator (ACC) register. Another example, they implemented some branch operation with an arithmetic operation in one instruction. These various options explain the large number of instructions compared to that in our design. In our design, various options of some instructions are implemented by checking only status register bits. For example, there are various ways of reading external data using the input register R_0 (i.e. serial/parallel input operations). These serial/parallel input options can be set easily by writing to the serial/parallel status bit in the STATUS register.

6.4 Cryptographic algorithms implementation results

This section compares the performance, and the optimality/operations efficiency results for different cryptographic algorithms implementations on programmable devices, i.e. processors, for passive RFID tags. This section gives an overview for the operations in each cryptographic algorithms and shows the area overhead due to programs written in the ROM memory or due to adding hardware extensions to perform some operation in hardware. It discusses also the achieved improvements in the performance and the optimality results for comparable cryptographic operations. At the end, it shows and explains some results for different message lengths for some cryptographic modules added to our design. A variety of different block and stream ciphers have been used in previous research efforts and in our work. Different ciphers offer different tradeoffs in security, area, and performance. In this section, we show that we can easily and efficiently incorporate different lightweight block and stream ciphers into our HSM. Comparing results between different ciphers does not enable a direct comparison of optimality, as a general trend, the resulting optimality for our HSM is significantly better than other research efforts using lightweight ciphers. For direct comparison of optimality results with other research efforts, we also incorporated AES into our HSM. We chose AES, because it is widely used in benchmarking, even though it is not lightweight and so might be a poor choice for most applications of EPC RFID systems.

Plos et al. analysed the block ciphers AES, SEA, Present and XTEA as well as the stream cipher Trivium. AES uses a substitution-permutation network (SPN) and works on a fixed block size of 128 bits and has a key length of 128 bits and it uses 10 rounds.

Present has been integrated in the ISO/IEC 29192-2 standard for lightweight cryptography in 2012. It uses a substitution-permutation network (SPN) and it supports key sizes of 80 and 128 bits and is applied on plain-text blocks with a length of 64 bits. Trivium is a hardware-oriented stream cipher. Trivium follows a very simple design strategy and allows to generate up to 2^{64} bits of key stream from an 80-bit initial value (IV) and an 80-bit secret key. Trivium operates on a 288-bit internal state, which needs to be initialized first before generation of the key stream is started. Table 6.4 compares some properties like key/message sizes and number of rounds of the selected cryptographic algorithms in Plos's work, Groß's work, and our work. Plos's implementations of AES-128, Present-80, and Trivium are faster than on 8-bit commercial micro-controller platforms (AVR [81] [26], PIC [45], 68HC08 [54], and 8051 [54] [70] micro-controllers). Except in the case of AES, their implementations are also the most compact ones. The encryption and decryption efficiency results are also much better compared to the other platforms[73].

Table 6.4: The selected cryptographic algorithms properties

Work	Algorithm	Cipher type	Key size (bits)	Block size (bits)	# of rounds
Plos[73]	AES	block	128	128	10
	Present	block	80	64	31
	Trivium	stream	80	-	128 for 128-bit message
Groß [42]	Present	block	80	64	31
	SEA	block	96	96	93
	XTEA	block	128	64	32
Our work	AES	block	128	128	10
	Simeck	block	64	32	32
	WG-5	stream	80	-	128 for 128-bit message
	ACE	AEAD	128	64	128

Groß et al. [42] choose the block ciphers SEA, Present and XTEA. They implemented software implementations without using hardware extensions (i.e. additional hardware units) and some other implementations using hardware executions. For example, Present-80 encryption/decryption operation consists of 31 substitution-permutation network (SPN)

rounds. The substitution layer of the SPN can be implemented with a single 4-bit S-box that is sequentially applied on the 16×4-bit block of the state. In the permutation layer, the state bits are mixed bitwise in a regular way. They implemented the substitution S-box operation and the permutation operation in both software and hardware. The hardware units are accessed using dedicated extended instructions.

In our work, we choose the block ciphers AES and Simeck, the stream cipher WG-5, and the Authenticated encryption ACE. Simeck combines the good design components from both SIMON and SPECK, in order to devise even more compact and efficient block ciphers. Simeck 32/64 has block size of 32 bits and key size of 64 bits. Each block encryption/decryption operation needs data and key loading at the beginning. The initialization phase including data/key loading is repeated with new data/key every block operation. The WG-5 cipher has been designed to produce keystream with guaranteed randomness properties such as: balance, long period, large and exact linear complexity, 3-level additive autocorrelation, and ideal 2-level multiplicative autocorrelation. It is resistant to Time/Memory/Data tradeoff attacks, algebraic attacks and correlation attacks. The WG-5 CIPHER has an 80-bit secret key and 80-bit Initialization vector. ACE aim to have a permutation that can achieve a balance between hardware cost and software efficiency for both hashing and AEAD functionalities. ACE is a 320-bit permutation and a generalization of sLiSCP [7] and sLiSCP-light [8] permutations with five 64-bit blocks.

6.4.1 Area considerations, ROM efficiency, and encryption/decryption efficiency

Table 6.5 is an example of the type of table used throughout Section 6.4 to compare results. The processor areas are not considered in efficiency calculations since the processor including the register file may be reused for other tasks such as other cryptographic operations or protocol handling tasks. The area used in efficiency calculations is referred in Table 6.5 as "additional area for cryptographic algorithms". In our implementation, this area includes both the ROM area and the extended cryptographic module area. But in Plos's design, this area includes only ROM area to implement such algorithm (i.e. GEs used in ROM to write a program for an algorithm). This area in case of Plos's results are generated using the synthesis tool after realizing into look-up tables. In case of our work, we used an estimated ROM area efficiency of 3.11 bits/GE to get estimated numbers of GEs for our programs. This ROM efficiency is the least used efficiency number compared to the other related work (i.e. more gate equivalents for same stored amount of bits).

Table 6.5: Encryption/decryption operations efficencies for some cryptographic algorithms

Work	Alg.	Code Size (bytes)	ROM Area (GEs)	Additional area for a cryptographic algorithm (GEs)	Msg. size	Enc. clock cycles	Enc. efficiency (10^{-7} ·bits/ GEs. cycles)	Dec. clock cycles	Dec. efficiency (10^{-7} · bits/ GEs. cycles)
Plos[73]	Present	920	1399	1399	64	28062	16.3	60427	7.6
	Trivium	332	754	754	128	85697	19.8	85697	19.8
	AES	1704	2911	2911	128	5064	86.8	8226	53.5
Our ASIP HSM	Simeck	44	114	888	64	107	6735.7	107	6735.7
	WG-5	16	42	1300	128	135	7293.4	135	7293.4
	AES	70	180	2385	128	249	2155.3	-	-
	ACE	34	86	4940	128	292	887.4	292	887.4

In Plos's design, the encryption/decryption execution times shown in Table 6.5 are considered for one block message in block ciphers (i.e. 128-bits for AES, and 64-bits for Present) and they are measured for generating 128-bit keystream for Trivium stream cipher. To be able to compare these results, the execution times for our work shown in Table 6.5 are calculated for same message sizes for similar chosen ciphers in Plos's work. i.e. 128 bits for Simeck block cipher, 64 bits for WG-5 stream cipher, 128 bits for AES block cipher, and 128 bits for ACE authenticated encryption module.

In order to determine the efficiency of an implementation, a message size in bits is divided by the product of execution time in clock cycles and additional area for a cryptographic algorithm in GEs. Table 6.5 shows that although our design uses additional hardware modules for performing cryptographic algorithms, it achieves very good efficiencies for both block and stream ciphers. The encryption performance for AES block cipher is improved over Plos's design by 27× and the encryption optimality is improved by 32×.

6.4.2 AES block cipher

AES is a standard cipher that is widely deployed to add security to many systems. However, it may not be the best option to be used for constrained devices as in EPC tags. The reason is that the relatively large power consumption of AES operations may not be suitable for extremely constrained devices as in EPC tags. We added AES for benchmarking reasons to show how effective our design is compared to other related work that use AES for security either in software or in hardware. We added a hardware instance of 8-bit Quark-AES implemented by Yu et. al. [94]. In Quark-AES architecture, every encryption function (AddRoundkey, SubBytes, ShiftRows, MixColumns) is performed serially, requiring only 16 clock cycles in a single AES round. The S-box used in Yu's implementation is same as Mathew's S-box [64]. Yu's AES implementation offers the benefits of low latency, single S-box instance, and no requirement to reorder inputs or outputs externally.

Table 6.6: AES Block encryption operation results in our ASIP HSM

	Standalone Yu's Quark-AES[94]	Quark-AES in our ASIP HSM
Block size (bits)	128	128
Key size (bits)	128	128
Program size (bytes)	-	70
Program size (GEs)	-	180
ROM area efficiency (bits/GE)	-	3.11
Processor area without memories & extensions (GEs)	-	4662
Cryptographic module Hardware cost (GEs)	2154	2205
Total chip area	2154	7047
Execution time for encryption operation (cycles)	216	249
Performance (bits/cycle)	0.59	0.51
Optimality score (10^{-7} bits/cycles·GE)	2751.1	729.5

Table 6.6 shows the encryption operation results in the standalone Yu's Quark-AES[94] and in our ASIP HSM after integrating an instance of Yu's AES into our ASIP processor. We didn't run decryption operations in our ASIP HSM. The wrapper for AES has been implemented using 51 GEs. The overall number of encryption code lines is 40 instructions (i.e. 70 bytes). We have only 33 clock cycles overhead to perform one block (128 bit) encryption operation in our ASIP HSM. This overhead is 15.3% of the encryption execution time for the standalone implementation. The optimality score has been decreased to 729.5 (10^{-7} bits/cycles·GE) for our ASIP processor while it is 2751.1 (10^{-7} bits/cycles·GE) for the standalone implementation. The reason of this decrease is the area overhead added because of the ASIP HSM itself (i.e. 4662 GEs) and the AES wrapper (i.e. 51 GEs) and the clock cycles overhead (i.e. 33 clock cycles) taken to operate Quark-AES by our ASIP processor.

We compare our results for Quark-AES cipher to the results in the other related work that have AES for security operations for EPC tags. These work include Plos's work[73], Ertl's work, and Fu's work. As discussed before, Plos's design is a RFID tag design that uses software-based approach. While Ertl's[30], and Fu's[39] designs are RFID tag designs that use specialized circuits approach. The hardware cost of implementing some AES block ciphers are shown in Table 6.7. The table compares the synthesis results for implementing these AES ciphers on our work, Plos's design work, and using other dedicated hardware AES modules. As mentioned before, the areas shown in Table 6.7 for our work includes both the area of the program written in the ROM utilized as a look-up table, and the extended hardware cryptographic modules. The area in Plos results considers only code size as cost factor. The code sizes in Table 6.7 don't include protocol or other codes. The additional hardware costs that are introduced by implementing cryptographic algorithms on our work and on Plos's processor are lower than by using stand-alone hardware module of Feldhofer [36].

Table 6.7: Synthesis results of some AES algorithm implementations

Algorithm	Code size (bytes)	ROM area efficiency (bits/GE)	ROM Area (GEs)	Total Area (GEs)
AES-128 - this work	70	3.11	180	2385
AES-128 - Plos [73]	1704	4.7	2911	2911
AES-128 - Feldhofer [36]	-	-	-	3400
AES-128 - Yu [94]	-	-	-	2154

In Plos's work [73], round keys are computed on-the-fly. Encryption and decryption operation were implemented. Decryption consumes significantly more execution time than encryption. S-box operation and inverse S-box operation are realized as look-up tables with 256 entries each. 39 registers are used by this implementation. The AES implementation with the area optimization target requires 5064 clock cycles for encryption and 8226 clock cycles for decryption. Code size of this version is 1704 bytes. As shown in Table 6.8, our ASIP has optimality score of 729.5 x 10^{-7} bits/cycles·GE for AES encryption operation while it is only 22.65 x 10^{-7} bits/cycles·GE in the case of Plos's work. The area (i.e. the

total area including cryptomodules areas and program code area) of both designs are relatively similar but our ASIP has much better results in the execution time for encryption operation. Our ASIP execution time is 4.9% of Plos's execution time for encryption operation. The main reason is that Plos's implementation for AES are fully done in software.

Ertl et. al. [30] use a low area implementation of the AES to provide mutual authentication based on a challenge-response protocol. The architecture of Ertl's AES module is based on the design of Feldhofer et al. presented in [36]. Ertl et. al. implemented only encryption functionality as it is only needed in the mutual authentication protocol. The datapath implements the basic AES operations, which are: SubBytes, MixColumns, AddRoundKey, and KeyScheduling. Their implementation uses an 8-bit architecture. The S-Box implementation is the biggest part of the datapath and is realized as combinational logic. The MixColumns implementation uses only one instead of four multipliers and processes one column in 28 clock cycles. A full encryption of one 128-bit block takes 1024 clock cycles.

The areas, showed in Table 6.8, for Ertl's designs consider the areas of AES specialized circuits module and the tag controller. Our optimality result (729.5 x 10^{-7} bits/cycles·GE) is almost quadruple the optimality score for the AES implemented in Ertl's design. Our AES area is nearly same as the area of the Ertl's AES. While our execution time is quarter the execution time of Ertl's design although Ertl's work use specialized circuits approach, the main reason is using a whole Quark-AES hardware module that has an efficient implementation targeting low-area and low execution time design goals.

Table 6.8: AES Block encryption operation results in the related work

Feature	Our ASIP HSM	Plos's work [73]	Ertl's work [30]	Fu's work [39]
Architecture	ASIP-based approach	Software-based approach	Specialized circuits approach	
Block size (bits)	128			
Key size (bits)	128			
Program size (bytes)	70	1704	-	-
Program size (GEs)	180	2911	-	-
ROM area efficiency (bits/GE)	3.11	4.7	-	-
Processor area without memories & extensions (GEs)	4662	5594	-	-
Controller unit area (GEs)	-	-	4300	27598
Cryptographic module Hardware cost (GEs)	2205	0	2770	4952
Total area	7047	8505	7070	32,550
Execution time for encryption operation (cycles)	249	5064	1024	204
Performance (bits/cycle)	0.514	0.019	0.125	0.627
Optimality score (10^{-7} bits/cycles·GE)	729.5	22.6	176.8	192.8

Fu et. al. [39] implemented AES engine as a part of a whole tag chip's baseband. The authors implemented both encryption and decryption functionalities in 4952 GEs. This area includes the four function modules for each round as well as the key expansion and the state schedule. The 128 bit encryption operation can be performed only in 204 clock cycles. This result makes optimality score of 192.8 x 10^{-7} bits/cycles·GE taking into consideration that the optimality score for Fu is calculated using the whole tag chip's baseband area.

As a summary, Table 6.8 shows different implementation results for AES for EPC tags using various design approaches. Using the same block and key sizes of 128-bit, our ASIP-

based HSM has the best optimality score of 729.5 x 10^{-7} bits/cycles·GE compared to 22.6 x 10^{-7} bits/cycles·GE optimality score for a software-besed design and 176.8 x 10^{-7} bits/cycles·GE, and 192.8 x 10^{-7} bits/cycles·GE optimality scores for specialized circuits-based designs. The key reason of these results is using an efficient AES hardware module as Quark-AES that targets both low area and low execution time design criteria. Although our HSM has an ASIP architecture, our HSM has the least overall area for AES for encryption operation compared to other related work.

6.4.3 Block ciphers: Present and Simeck

The hardware cost of implementing some block ciphers are shown in Table 6.9. The table compares the synthesis results for implementing these ciphers on our work, Plos's design work, and using another specialised circuits Present module. As mentioned before, the areas shown in Table 6.9 for our work includes both the area of the program written in the ROM utilized as a look-up table, and the extended hardware cryptographic modules. The area in Plos results considers only code size as cost factor. The code sizes in Table 6.9 don't include protocol or other codes. We used an implementation of Simeck developed by Yang et al.[91].

Table 6.9: Synthesis results of some block ciphers implementations

Algorithm	Code size (bytes)	ROM area efficiency (bits/GE)	ROM Area (GEs)	Total Area (GEs)
Simeck - this work	44	3.11	114	888
Present-80 Plos [73]	920	5.3	1399	1399
Present-80 Poschmann [75]	-	-	-	1075

Table 6.10 shows the impact of using hardware extensions on the program size and the execution time of Encryption/decryption operations using block ciphers. The overall chip area, the program size, and the execution time for Present-80 block cipher implementation have been reduced after using hardware extensions in Groß's work. The overall chip area (in GEs) includes the micro-controller, the hardware extensions, and the ROM

memory (realised in GEs). The Performance as well as the optimality scores have been enhanced after using hardware extensions. In our work, we use Simeck 32/64 block cipher and we choose similar message size of 64 bits (two data blocks) to compare the execution time, the performance, and the optimality result with Groß's work. As shown in Table 6.10, our design has a major advantage regarding low execution time and high data rate performance for encryption/decryption operations comparing to Groß's work results. For example, the execution time for encrypting 64-bit data message is 107 clock cycles in our work compared to 26,062 and 11,469 clock cycles for Groß's work. As a result of that, the data rate performance in our design is much better than that in Groß's design although the total chip areas are relatively same. We have 0.6 bits/cycle data rate while it is 0.0015 bits/cycle in Groß's design with no extension implementation and it is 0.0024 bits/cycle in Groß's design with s-box and permutation hardware extensions implementation.

Table 6.10: Block encryption/decryption operation results

	Plos's work [73]	Groß's work[42]	This work
Algorithm	Present block encryption/decryption		Simeck block encryption/decryption
HW Extensions	no extensions	Sbox+Perm	Simeck cipher
Message size (bits)	64	64	64
Block size (bits)	64	64	32
Key size (bits)	80	80	64
Program size (bytes)	944	324	44
Program size (GEs)	1525	833	114
ROM area efficiency (bits/GE)	4.96	3.11	3.11
Processor area without memories & extensions (GEs)	4300	4300	4662
Hardware extension cost (GEs)	0	411	774
Total chip area	5825	5544	5550
Execution time for encryption operation (cycles)	28062	11468	107
Execution time for decryption operation (cycles)	60426	42479	107
Performance (bits/cycle)	0.0014	0.0024	0.5981
Optimality score (10^{-7} bits/cycles·GE)	2.48	4.28	1077.71

The other important advantage in our design is the low program size. The program for encrypting/decrypting 64 bits has 44 bytes compared to 944 bytes and 324 bytes in Groß's work implementations. The main reason for that is using a complete hardware ciphers like Simeck and WG-5. Now, cryptographic operations are performed completely in hardware

instead of software, as in Groß's implementations, which is much slower and consumes more program size as shown in Table 6.10. The overall chip area includes the processor without extensions and the hardware extension and the program memory (in GEs). We use an estimated ROM efficiency number to get the estimated realization for the program ROM in Gate Equivalents (GEs). We use the least ROM efficiency number 3.11 in Groß's work. The overall chip area for both implementations that use extensions are almost same.

The optimality score is a measure for how efficient is the design with regards to the chip area, and the execution time for certain operation. It is clearly shown that our design has better optimality score of 1,077.7 (10^{-7} bits/cycles·GE) compared to 2.48 and 4.28 (10^{-7} bits/cycles·GE) in Groß's implementations. The calculated optimality score for Groß's design are for both encryption and decryption operations. The used execution time is the average execution time for one encryption and one decryption operations for 64 bits message size. In general, the Performance as well as the optimality scores have been enhanced. The performance for block cipher operations has been improved by 252.8× and 414.9× compared to Groß's work implementations results and the optimality score has been improved 251.8× and 434× compared to Groß's work implementations results.

Table 6.11 shows the encryption/decryption operation results for different message sizes for Simeck 32/64 block cipher on our ASIP HSM. Simeck 32/64 has block size of 32 bits and key size of 64 bits. Each block encryption/decryption operation needs data and key loading at the beginning. The initialization phase including data/key loading is repeated with new data/key every block operation. For message size of 32 bits, one block encryption or decryption is needed. The program size in this case is 39 bytes. For message size of 64, 128, or 256 bits, more than one block encryption or decryption is needed. The program in this case uses loops to iterate the same number of message blocks (i.e. message size divided by block size). The program size in this case is 44 bytes. As shown in Table 6.11, the performance results for different message sizes are almost same (i.e. around 0.6). The slight difference comes from the overhead from looping certain times depending on the message size. The optimality score varies from 1055.5 to 1111.4 (10^{-7} bits/cycle·GE).

Table 6.11: Encryption/decryption operation results using Simeck 32/64 on our ASIP HSM

Operation	Feature	Simeck 32/64 cipher [91]			
Initialization and key/data loading in each iteration	Message size (bits)	32	64	128	256
	Program size (bytes)	37			
	Program size (GEs)	96			
	Execution time (cycles)	44			
Encryption/ decryption operation	Program size (bytes)	39	44		
	Program size (GEs)	101	114		
	ROM area efficiency (bits/GE) (estimated)	3.11			
	Simeck cipher area (GEs)	774			
	HSM area (GEs)	4662			
	Overall area (GEs)	5537	5550		
	Execution time for encryption/decryption operation (cycles)	52	107	217	437
	Encryption/decryption performance (bits/cycle)	0.62	0.60	0.59	0.59
	Optimality score (10^{-7} bits/cycle·GE)	1111.4	1077.7	1062.8	1055.5

6.4.4 Stream ciphers: Trivium and WG-5

The hardware cost of implementing some stream ciphers are shown in Table 6.12. The table compares the synthesis results for implementing these ciphers on our work, Plos's design work, and using another specialised circuits Trivium module. The areas shown in Table 6.12 for our work includes both the area of the program written in the ROM utilized as a look-up table, and the extended hardware cryptographic modules. The area in Plos results considers only code size as cost factor. The code sizes in Table 6.12 don't include

protocol or other codes. We used an implementation of WG-5 developed by Aagaard et al.[5].

Table 6.12: Synthesis results of some stream ciphers implementations

Algorithm	Code size (bytes)	ROM area efficiency (bits/GE)	ROM Area (GEs)	Total Area (GEs)
WG-5 - this work	16	3.11	42	1300
Trivium Plos [73]	332	3.6	754	754
Trivium Feldhofer [35]	-	-	-	2390

The general trend for stream cipher results is close to that in block cipher results. Table 6.13 shows software-approach implementation results for Trivium stream cipher using Plos's micro-controller. Groß et al. have not implemented stream ciphers in their work. Table 6.13 shows the implementation results for WG-5 stream cipher using our ASIP processor and compares these results to Trivium stream cipher results using Plos's micro-controller. As in block cipher results, our design has some advantages with regards to the low execution time and the low program size. The execution time to encrypt stream of 128 bits is 135 clock cycles in our design while it takes 85,697 clock cycles to encrypt the same message using Trivium cipher on Plos's design. These results reflect to the bit rate performance result which is 0.948 bits/cycle in our design but it is 2.689x10^{-3} bits/cycle in Plos's design. In other words, the bit rate has been improved over Plos's design by 352.5$\times$ for generating 128-bit key-stream.

Table 6.13: Stream encryption/decryption operation results

	Plos's work[73]	This work
Algorithm	Trivium stream encryption/decryption	WG-5 stream encryption/decryption
HW Extensions	no extensions	WG-5 cipher
Message size (bits)	128	128
Program size (bytes)	332	16
Program size (GEs)	754	42
ROM area efficiency (bits/GE)	3.52	3.11
Processor area without memories & extensions (GEs)	4300	4662
Hardware extension cost (GEs)	0	1258
Total chip area	5054	5962
Execution time for encryption operation (cycles)	85697	135
Execution time for decryption operation (cycles)	85697	135
Performance (bits/cycle)	$1.494\text{x}10^{-3}$	0.948
Optimality score (10^{-7} bits/cycles·GE)	2.96	1590.32

The overall chip area in our design is much higher than the overall chip area in Plos's design. The base processors without extensions and memories have close area utilization. But our design includes hardware extension for WG-5 cipher. The WG-5 cipher area is 1258 GEs. This explains the difference in area (908 GEs) between the both designs. Our HSM achieves much higher performance because the cipher is implemented in hardware, rather than software.

Table 6.11 shows the encryption/decryption operation results for different message sizes for WG-5 stream cipher on our ASIP HSM. The initialization phase in WG-5 cipher is done only one time at the beginning. Hence, the overhead resulted from the initialization phase affects more the messages that have short lengths. For message size of 16 bits, the encryption/decryption performance is 0.696 bits/cycle while it is 0.901, 0.948, and 0.973 bits/cycle for messages have lengths of 64, 128, 256 bits respectively. As shown in Table 6.14, the optimality score varies between 1,166.9 to 1,632.6 (10^{-7} bits/cycle·GE) for the shown messages lengths.

Table 6.14: Encryption/decryption operation results using WG-5 on our ASIP HSM

Operation	Feature	WG-5 cipher [5]			
Initialization	Message size (bits)	16	64	128	256
	Program size (bytes)	191			
	Program size (GEs)	492			
	Execution time (cycles)	271			
Encryption/ decryption operation	Program size (bytes)	16	16	16	16
	Program size (GEs)	42	42	42	42
	ROM area efficiency (bits/GE) (estimated)	3.11	3.11	3.11	3.11
	WG-5 cipher area (GEs)	1258	1258	1258	1258
	HSM area (GEs)	4662	4662	4662	4662
	Overall area (GEs)	5,962	5,962	5,962	5,962
	Execution time for encryption/decryption operation (cycles)	23	71	135	263
	Encryption/decryption performance (bits/cycle)	0.696	0.901	0.948	0.973
	Optimality score (10^{-7} bits/cycle·GE)	1,166.9	1,511.9	1,590.3	1,632.6

6.4.5 ACE authenticated encryption module

ACE authenticated encryption module has quite complicated operations compared to WG-5 and Simeck ciphers. The data width for inputs/outputs are 64 bit and there are four

mode of operations to support authenticated encryption operations. We used the ACE implementation developed by Aagaard et al. [4]. Table 6.16 shows the execution times for different operations of ACE authenticated encryption module using our ASIP HSM. Our ASIP HSM consumes 725 clock cycles for the loading/initialization phase while the standalone ace module consumes 651 clock cycles. The main reason for this difference comes from loading immediate values to Ace module. Loading 32 bits consumes 3 clock cycles in our ASIP HSM and another clock cycle to load this data from a register to ACE inputs. There are 10 32-bit loading operations during the ace load phase. Initialization phase and processing of associated data consume 516 clock cycles on standalone module while they consume 539 clock cycles using our ASIP HSM. Our ASIP HSM has additional 15 cycles for encryption phase preparing. The detailed distribution for the execution time in loading/initialization phase is shown in Table 6.15. The overall loading/initialization time for ACE in our ASIP HSM is 725 clock cycles. This number is between the overall loading/initialization times for standalone ace with 32-bit and 64-bit data interfaces. Our ASIP HSM uses an ACE module with 64-bit data interface.

Table 6.15: Loading/initialization execution time for ACE in our ASIP HSM

Operation	**Standalone ace module with 32-bit data interface[4]**	**Standalone ace module with 64-bit data interface[4]**	**Using our ASIP HSM**	
	execution time (cycles)		execution time (cycles)	# instructions
Loading	409	135	171	43
Initialization	516	516	539	32
Encryption preparation	-	-	15	15
Overall loading/ initialization	925	651	725	90

Table 6.16 shows the comparison of the performance results for ACE encryption/ decryption operations in our ASIP HSM and in a standalone module. The execution time

Table 6.16: Encryption/decryption operation results for ACE in our ASIP HSM

Msg. size (bits)	64		128		256		512	
	standalone [4]	ASIP HSM	standalone [4]	ASIP HSM	standalone [4]	ASIP HSM	standalone [4]	ASIP HSM
Execution time (cycles)	129	146	258	292	516	584	1032	1168
Performance (bits/cycle)	0.0821	0.0735	0.1408	0.1259	0.2194	0.1956	0.3042	0.2705
Optimality score ($10^{-}7$ bits/ cycles·GE)	185.01	75.85	317.51	129.91	494.62	201.87	685.95	279.18

in our ASIP HSM has 17 clock cycles overhead for encrypting/decrypting 64 bits. This overhead comes from loading data from memory and loading 64 bits to data input of ACE module and reading 64 bits from data output of ACE module. Our ASIP HSM interacts with ACE module through 7 extended instruction options as shown in Table 6.17. Our general interface for extended modules allows loading two 16-bit input data and reading from 16-bit output data. The data unit for ACE module is 64-bit so 64 bits data can be loaded in two clock cycles and 64 bits data can be read in four clock cycles using the ace instruction options. One encryption permutation cycle requires 16 steps and each step requires 8 rounds and a round can run in one clock cycle. This operation is done in our ASIP HSM by executing `ace.run` instruction for 128 clock cycles using the accelerated mode.

The performance results shown in Table 6.16 consider the overall load/initialization time into their calculations. The load/initialization operation is done only one time. As message size gets increased, performance has better results. For message size of 512 bits, our ASIP has 0.2705 bits/cycle performance while it is 0.3042 bits/cycle for standalone ace module. The reason is the overhead in the encryption/decryption time beside the overhead in the loading/initialization phase.

Table 6.17: ACE instruction options in our ASIP HSM

Ace instruction option	Meaning
`ace.ldm`	load mode of operation and control bits
`ace.ldl`	load low 32-bits
`ace.ldh`	load high 32-bits and read word0
`ace.rdw1`	read word1
`ace.rdw2`	read word2
`ace.rdw3`	read word3
`ace.run`	run

The optimality score results for our ASIP HSM are less than half of the optimality score results for the standalone module. The optimality score calculations take into consideration only the standalone ace module area (i.e. 9688 GEs) while the optimality score calculations for our work take into consideration the areas of ace module with its wrapper, our ASIP HSM, and the program memory (i.e. 9688 GEs) which are more than double of the ace module area.

The area details for ace module in our ASIP are shown in Table 6.18. The standalone ace has 4435 GEs while the ace module with the wrapper is 4940 GEs. The wrapper itself is 505 GEs which is quite large (11.39% of ace module area). The main reason for that is using flip-flops either for loading the data input or reading from the data output in multiple cycles.

Table 6.18: Area results for ACE using our ASIP HSM

Module	Area (GEs)	
ASIP HSM (GEs)	4662	
Ace module	stand-alone[4]	with wrapper
	4435	4940
ROM Operation	Initialization	Encryption/ decryption
# instructions	90	19
ROM Area (bytes)	156	34
ROM Area (GEs)	406	86

Chapter 7

Conclusion and Future Work

7.1 Conclusion

In this thesis, we presented an optimized application-specific instruction set processor (ASIP) for an ultralight Hardware Security Module (HSM). We choose EPC tags as a prototype for ultralight devices. HSMs are computing devices that protect cryptographic keys and provide cryptographic operations. The HSM combines all security-related functions for EPC tag. The goal of this research is to demonstrate that using an ASIP architecture for ultralight HSMs provides benefits in terms of trade-offs between flexibility, extensibility, and efficiency.

The current commercial RFID tag designs use specialised hardware circuits approach. This approach can achieve the lowest area and power consumption; however, it lacks flexibility and takes long time to develop. On the contrary, the software-based approach is used in a few academic works. These designs are much slower and consume more energy. The main advantages of using software-based approach is the high flexibility and the shortest time to market. The ASIP-based approach that is used in our work can achieve balance between the rich functionality, the low cost chips, the high performance, and the reasonable security requirements as well as high level of flexibility and the relatively short time to market.

The instruction set and the micro-architecture of the ASIP processor are chosen such that the overall chip area including the program ROM size is small as possible and the

processor's operations can be run efficiently (i.e. in less time). The chosen instruction length of 14-bits covers all the supported instructions. It is important to keep the instruction length as small as possible because this length affects the program ROM size. The instruction length depends on the number of instruction fields and their sizes, the number of operations, and the number of supported addressing modes. 14-bits are the least number that can be used for the instruction length to cover all supported instructions. Three instruction formats are used in our ASIP: Register-type (R-type), Immediate-1-type (I_1-type), Immediate-2-type (I_2-type).

Our ASIP processor uses reduced instruction set computing (RISC) architecture that has fixed instruction length of 14-bits, three instruction formats, and a single-cycle instruction implementation. The ASIP processor is able to perform general operations as well as special operations using the added hardware units. The added units can be integrated in easy manner as plug-in units. This is what we call "flexibility" in this thesis. The added units can be accessed by custom instructions defined by the system designer at the design level phase (i.e. before fabrication), this adds more functionalities to the ASIP processor by only integrating the required units into the ASIP's datapath.

The proposed ASIP architecture for EPC tags can reduce the unit cost. Compared to specialised hardware approach, major part of the design effort in the ASIP-based approach is moved from the hardware (i.e. relatively expensive) to the software (cheap). This reduces the non-recurring cost (NRE). The ASIP-based approach allows large scale production by combining more than one tag model in one fabrication process.

Beside performing cryptographic functions such as: random number generation, data encryption/decryption, and message authentication, our design supports some hardware security mechanisms to protect sensitive data in the memory or in the register file. These mechanisms are designed to be suitable for extremely constrained devices. These mechanisms include control access techniques for the memory, triple-mode redundancy (TMR) fault-tolerant technique for some special-purpose registers, secure/non-secure states, and memory passwords. The programmability in our design allows for adding more security services using the available hardware resources. These services may include entity or mutual authentication, keys/passwords management, and temporarily tag lock. Our design is the first hardware security module for EPC RFID tags that performs cryptographic functions as well as supports some hardware/software mechanisms to safeguard sensitive data and private keys.

Our ASIP processor can be implemented with 4,662 gate equivalent units (GEs) for 65 nm CMOS technology excluding cryptographic units and memories. We integrated and analysed four cryptographic modules: AES block cipher, Simeck block cipher, WG-5 stream cipher, and ACE authenticated encryption module. Our HSM achieves very good efficiencies for both block and stream ciphers. Specifically for the AES cipher, we improve over a previous programmable AES implementation result by 32×. We increase performance dramatically and increase/decrease area by 17.97/17.14% respectively. Our ASIP instruction set provides customizable instruction for different ciphers and our microarchitecture provides a flexible but standardized interface to cryptographic modules. The cost of incorporating a cryptographic module into our HSM ranges from an area overhead of 2.37% and 40 lines of assembly code for the AES cipher to 10.39% overhead and 109 lines of code for the ACE authenticated cipher. These results fulfill the requirements of extremely constrained devices as in EPC tags and allow the inclusion of cryptographic units into the datapath of our ASIP processor.

7.2 Future works

7.2.1 Implementation of more security services

One of the advantages of our work is the ability to implement different cryptographic services in easy way. In our work, we implemented some security services such as random number generation, input/output data encryption/decryption, memory data encryption/decryption, and message authentication. Some services may be required to be implemented, depending on the used application, such as mutual authentication, key management. The important note here that these different cryptographic services can use same cryptographic units. For example, a stream cipher can be used to implement random number generation and input/output data encryption/decryption. The main advantage of using software in our work is the ability to manage the added cryptographic modules to implement different services using minimal design efforts.

7.2.2 Security validations

Hardware security modules have to be validated for security by one of the security evaluations like FIPS 140-2 standard or Common Criteria (CC). Usually FIPS 140-2 validation

process has to be handled by independent third-party laboratories that are accredited by National Voluntary Laboratory Accreditation Program (NVLAP) at NIST. It is beneficial to perform this validation process ourselves for our HSM to check that our design meets the FIPS 140-2 guidelines and requirements taking in mind that our HSM is not a commercial product. Conformance to FIPS 140-2 standard is a necessary step toward maintaining the intended security level. We have done some validations to FIPS 140-2 and most of our design specifications meet the security requirements for level 3. More works are required to meet the security requirements for key management and mitigation of some attacks such as side channel attacks.

7.2.3 The communication module architecture

In our EPC tag design, we divided the tag digital baseband into two modules: the communication module (CM), and the hardware security module (HSM). We want to explore different architectures for the CM. To get the lowest possible area for the CM, the CM has to be designed using the specialized circuit approach. This approach provides low level of flexibility or programmability for the CM. One promising idea is to use the ASIP-based approach. The ASIP-based approach combines the benefits of the flexibility and the low area, low power consumption, and high performance at the same time.

References

[1] The common criteria. https://www.commoncriteriaportal.org/. Accessed: 2021-01-23.

[2] Sae j3101 hardware protected security for ground vehicles. http://saemobilus.sae.org/content/J3101_202002. Accessed: 2021-01-23.

[3] Globalplatform based trusted execution environment and trustzone ready. *Arm.com*, 2013.

[4] Mark Aagaard, Riham AlTawy, Guang Gong, Kalikinkar Mandal, and Raghvendra Rohit. Ace: An authenticated encryption and hash algorithm. *Submission to NIST-LWC*, 2019.

[5] Mark D Aagaard, Guang Gong, and Rajesh K Mota. Hardware implementations of the wg-5 cipher for passive rfid tags. In *2013 IEEE International Symposium on Hardware-Oriented Security and Trust (HOST)*, pages 29–34. IEEE, 2013.

[6] Himja Agrawal and PR Badadapure. A survey paper on elliptic curve cryptography. *International Research Journal of Engineering and Technology*, (04), 2016.

[7] Riham AlTawy, Raghvendra Rohit, Morgan He, Kalikinkar Mandal, Gangqiang Yang, and Guang Gong. sliscp: Simeck-based permutations for lightweight sponge cryptographic primitives. In *International Conference on Selected Areas in Cryptography*, pages 129–150. Springer, 2017.

[8] Riham Altawy, Raghvendra Rohit, Morgan He, Kalikinkar Mandal, Gangqiang Yang, and Guang Gong. Sliscp-light: Towards hardware optimized sponge-specific cryptographic permutations. *ACM Transactions on Embedded Computing Systems (TECS)*, 17(4):1–26, 2018.

[9] Alex Arbit, Yoel Livne, Yossef Oren, and Avishai Wool. Implementing public-key cryptography on passive rfid tags is practical. *International Journal of Information Security*, 14(1):85–99, 2015.

[10] Frederik Armknecht, Matthias Hamann, and Vasily Mikhalev. Lightweight authentication protocols on ultra-constrained rfids-myths and facts. In *International Workshop on Radio Frequency Identification: Security and Privacy Issues*, pages 1–18. Springer, 2015.

[11] Jean-Philippe Aumasson, Luca Henzen, Willi Meier, and María Naya-Plasencia. Quark: A lightweight hash. In *CHES*, volume 6225, pages 1–15. Springer, 2010.

[12] Henri Barthel. EPCglobal–RFID standards & regulations. 2005.

[13] Lejla Batina, Jorge Guajardo, Tim Kerins, Nele Mentens, Pim Tuyls, and Ingrid Verbauwhede. Public-key cryptography for RFID-tags. In *Pervasive Computing and Communications Workshops, 2007. PerCom Workshops' 07. Fifth Annual IEEE International Conference on*, pages 217–222. IEEE, 2007.

[14] R Beaulieu, D Shors, J Smith, S Treatman-Clark, B Weeks, and L Wingers. The SIMON and SPECK Families of Lightweight Block Ciphers. Cryptology ePrint Archive, Report 2013/404, 2013.

[15] Karthikeyan Bhargavan and Gaëtan Leurent. On the practical (in-) security of 64-bit block ciphers: Collision attacks on HTTP over TLS and OpenVPN. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 456–467. ACM, 2016.

[16] Andrey Bogdanov, Lars R Knudsen, Gregor Leander, Christof Paar, Axel Poschmann, Matthew JB Robshaw, Yannick Seurin, and Charlotte Vikkelsoe. PRESENT: An ultra-lightweight block cipher. In *CHES*, volume 4727, pages 450–466. Springer, 2007.

[17] Andrey Bogdanov, Gregor Leander, Christof Paar, Axel Poschmann, Matt Robshaw, and Yannick Seurin. Hash functions and RFID tags: Mind the gap. *Cryptographic Hardware and Embedded Systems–CHES 2008*, pages 283–299, 2008.

[18] Duy-Hieu Bui, Diego Puschini, Simone Bacles-Min, Edith Beigné, and Xuan-Tu Tran. Ultra low-power and low-energy 32-bit datapath aes architecture for iot applications. In *2016 International Conference on IC Design and Technology (ICICDT)*, pages 1–4. IEEE, 2016.

[19] C De Canniere. Trivium specifications. *http://www. ecrypt. eu. org/stream/p3ciphers/trivium/trivium_p3. pdf*, 2005.

[20] David Canright. A very compact s-box for aes. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 441–455. Springer, 2005.

[21] Hee-Jin Chae, Mastooreh Salajegheh, Daniel J Yeager, Joshua R Smith, and Kevin Fu. Maximalist cryptography and computation on the wisp uhf rfid tag. In *Wirelessly Powered Sensor Networks and Computational RFID*, pages 175–187. Springer, 2013.

[22] Joan Daemen and Vincent Rijmen. Aes proposal: Rijndael. 1999.

[23] Raghu Das and Peter Harrop. Rfid forecasts, players and opportunities 2009–2019. *IDTechEx report*, 2009.

[24] Dang Nguyen Duc, Hyunrok Lee, and Kwangjo Kim. Enhancing security of EPC-global gen-2 RFID against traceability and cloning. *Auto-ID Labs Information and Communication University, White Paper*, 2006.

[25] Morris J Dworkin. Sha-3 standard: Permutation-based hash and extendable-output functions. 2015.

[26] Thomas Eisenbarth, Sandeep Kumar, Christof Paar, Axel Poschmann, and Leif Uhsadel. A survey of lightweight-cryptography implementations. *IEEE Design & Test of Computers*, 24(6):522–533, 2007.

[27] Daniel Engels, Xinxin Fan, Guang Gong, Honggang Hu, and Eric M Smith. Ultra-lightweight cryptography for low-cost RFID tags: Hummingbird algorithm and protocol. *Centre for Applied Cryptographic Research (CACR) Technical Reports*, 29, 2009.

[28] Daniel Engels, Xinxin Fan, Guang Gong, Honggang Hu, and Eric M Smith. Hummingbird: ultra-lightweight cryptography for resource-constrained devices. In *International Conference on Financial Cryptography and Data Security*, pages 3–18. Springer, 2010.

[29] EPCradio-frequencyidentityprotocolsclass EPCGlobal and K Chiew. Radio-frequency identity protocols class-1 generation-2 uhf rfid protocol for communications at 860 mhz–960 mhz version 1.0. 9. *K. Chiew et al./On False Authenticationsfor C1G2 Passive RFID Tags*, 65, 2004.

[30] Johann Ertl, Thomas Plos, Martin Feldhofer, Norbert Felber, and Luca Henzen. A security-enhanced uhf rfid tag chip. In *2013 Euromicro Conference on Digital System Design*, pages 705–712. IEEE, 2013.

[31] R Escherich, I Ledendecker, C Schmal, B Kuhls, C Grothe, and F Scharberth. She–secure hardware extension–functional specification version 1.1. *Hersteller Initiative Software (HIS) AK Security*, 2009.

[32] Xinxin Fan, Kalikinkar Mandal, and Guang Gong. WG-8: A lightweight stream cipher for resource-constrained smart devices. In *International Conference on Heterogeneous Networking for Quality, Reliability, Security and Robustness*, pages 617–632. Springer, 2013.

[33] Horst Feistel. Cryptography and computer privacy. *Scientific american*, 228(5):15–23, 1973.

[34] Martin Feldhofer, Sandra Dominikus, and Johannes Wolkerstorfer. Strong authentication for RFID systems using the AES algorithm. In *CHES*, volume 4, pages 357–370. Springer, 2004.

[35] Martin Feldhofer and Johannes Wolkerstorfer. Hardware implementation of symmetric algorithms for rfid security. In *RFID security*, pages 373–415. Springer, 2008.

[36] Martin Feldhofer, Johannes Wolkerstorfer, and Vincent Rijmen. Aes implementation on a grain of sand. *IEE Proceedings-Information Security*, 152(1):13–20, 2005.

[37] NIST FIPS. 140-3,"security requirements for cryptographic modules," 2019. *US Department of Commerce/National Institute of Standards and Technology.*

[38] PUB FIPS. 180-2 federal information processing standards publication. *SECURE HASH STANDARD, National Institute of Standards and Technology*, 2002.

[39] Lingzhi Fu, Xiang Shen, Linghao Zhu, and Junyu Wang. A low-cost uhf rfid tag chip with aes cryptography engine. *Security and Communication Networks*, 7(2):365–375, 2014.

[40] EPC Global. Epc radio-frequency identity protocols generation-2, uhf rfid standard, specification for rfid air interface protocol for, communications at 860 mhz-960 mhz, 2018.

[41] Gyõzõ Gódor, Norbert Giczi, and Sándor Imre. Elliptic curve cryptography based mutual authentication protocol for low computational complexity environment. In *Wireless Pervasive Computing (ISWPC), 2010 5th IEEE International Symposium on*, pages 331–336. IEEE, 2010.

[42] Hannes Groß and Thomas Plos. On using instruction-set extensions for minimizing the hardware-implementation costs of symmetric-key algorithms on a low-resource microcontroller. In *International Workshop on Radio Frequency Identification: Security and Privacy Issues*, pages 149–164. Springer, 2012.

[43] OMTP Hardware Working Group et al. Omtp hardware requirements and defragmentation. *Trusted Environment OMTP TR1 v1.1. Open Mobile Terminal Platform*, 2009.

[44] Trusted Computing Group. Tpm library specification 2.0. 2014.

[45] Caio Gubel. Advanced encryption standard using the pic16xxx, 2002.

[46] Bao Guihao, Zhang Minggao, Liu Jiuwen, and Li Yin. The design of an RFID security protocol based on RSA signature for e-ticket. In *Information Management and Engineering (ICIME), 2010 The 2nd IEEE International Conference on*, pages 636–639. IEEE, 2010.

[47] Jian Guo, Thomas Peyrin, and Axel Poschmann. The photon family of lightweight hash functions. In *Annual Cryptology Conference*, pages 222–239. Springer, 2011.

[48] Panu Hamalainen, Timo Alho, Marko Hannikainen, and Timo D Hamalainen. Design and implementation of low-area and low-power AES encryption hardware core. In *Digital System Design: Architectures, Methods and Tools, 2006. DSD 2006. 9th EUROMICRO Conference on*, pages 577–583. IEEE, 2006.

[49] George Hatzivasilis, Konstantinos Fysarakis, Ioannis Papaefstathiou, and Charalampos Manifavas. A review of lightweight block ciphers. *Journal of cryptographic Engineering*, 8(2):141–184, 2018.

[50] Martin Hell, Thomas Johansson, and Willi Meier. Grain: a stream cipher for constrained environments. *International Journal of Wireless and Mobile Computing*, 2(1):86–93, 2007.

[51] Walter Hinz, Klaus Finkenzeller, and Martin Seysen. Secure UHF Tags with Strong Cryptography.

[52] Payment Card Industry. Data security standard. *Requirements and Security Assessment version*, 3:58, 2018.

[53] Geneva ISO. Information technology–trusted platform module. 2009.

[54] Abdellatif JarJar. Improvement of feistel method and the new encryption scheme. *Optik*, 157:1319–1324, 2018.

[55] Ari Juels. RFID security and privacy: A research survey. *IEEE journal on selected areas in communications*, 24(2):381–394, 2006.

[56] Kuljeet Kaur, Neeraj Kumar, Mukesh Singh, and Mohammad S Obaidat. Lightweight authentication protocol for rfid-enabled systems based on ecc. In *2016 IEEE Global Communications Conference (GLOBECOM)*, pages 1–6. IEEE, 2016.

[57] Elif Bilge Kavun and Tolga Yalcin. A lightweight implementation of keccak hash function for radio-frequency identification applications. In *International Workshop on Radio Frequency Identification: Security and Privacy Issues*, pages 258–269. Springer, 2010.

[58] Cheol-Joong Kim, Sung-Yeol Yun, and Seok-Cheon Park. A lightweight ECC algorithm for mobile RFID service. In *Ubiquitous Information Technologies and Applications (CUTE), 2010 Proceedings of the 5th International Conference on*, pages 1–6. IEEE, 2010.

[59] Jia Hao Kong, Li-Minn Ang, and Kah Phooi Seng. A comprehensive survey of modern symmetric cryptographic solutions for resource constrained environments. *Journal of Network and Computer Applications*, 49:15–50, 2015.

[60] Xuejia Lai and James L Massey. A proposal for a new block encryption standard. In *Workshop on the Theory and Application of of Cryptographic Techniques*, pages 389–404. Springer, 1990.

[61] Jong-Wook Lee, Ngoc Dang Phan, Duong Huynh-Thai Vo, and Vinh-Hao Duong. A fully integrated EPC Gen-2 UHF-band passive tag IC using an efficient power management technique. *IEEE Transactions on Industrial Electronics*, 61(6):2922–2932, 2014.

[62] Yiyuan Luo, Qi Chai, Guang Gong, and Xuejia Lai. A lightweight stream cipher WG-7 for RFID encryption and authentication. In *Global Telecommunications Conference (GLOBECOM 2010), 2010 IEEE*, pages 1–6. IEEE, 2010.

[63] Adam SW Man, Edward S Zhang, Vincent KN Lau, Chi Ying Tsui, and Howard C Luong. Low power vlsi design for a rfid passive tag baseband system enhanced with an aes cryptography engine. In *2007 1st Annual RFID Eurasia*, pages 1–6. IEEE, 2007.

[64] Sanu Mathew, Sudhir Satpathy, Vikram Suresh, Mark Anders, Himanshu Kaul, Amit Agarwal, Steven Hsu, Gregory Chen, and Ram Krishnamurthy. 340 mv–1.1 v, 289 gbps/w, 2090-gate nanoaes hardware accelerator with area-optimized encrypt/decrypt gf (2 4) 2 polynomials in 22 nm tri-gate cmos. *IEEE Journal of Solid-State Circuits*, 50(4):1048–1058, 2015.

[65] Amir Moradi, Axel Poschmann, San Ling, Christof Paar, and Huaxiong Wang. Pushing the limits: a very compact and a threshold implementation of AES. In *Eurocrypt*, volume 6632, pages 69–88. Springer, 2011.

[66] Yassir Nawaz and Guang Gong. The wg stream cipher. *ECRYPT Stream Cipher Project Report 2005*, 33, 2005.

[67] Yassir Nawaz and Guang Gong. Wg: A family of stream ciphers with designed randomness properties. *Information Sciences*, 178(7):1903–1916, 2008.

[68] Haifeng Niu, Eyad Taqieddin, and Sarangapani Jagannathan. Epc gen2v2 rfid standard authentication and ownership management protocol. *IEEE Transactions on Mobile Computing*, 15(1):137–149, 2015.

[69] Liaojun Pang, Liwei He, Qingqi Pei, and Yumin Wang. Secure and efficient mutual authentication protocol for RFID conforming to the EPC C-1 G-2 standard. In *Wireless communications and networking conference (WCNC), 2013 IEEE*, pages 1870–1875. IEEE, 2013.

[70] Marko Pavlin. Encription using low cost microcontrollers. In *42nd International Conference on Microelectronics, Devices and Materials and the Workshop on MEMS and NEMS, Society for Microelectronics Electronic*, pages 189–194. Citeseer, 2006.

[71] Pedro Peris-Lopez, Julio Cesar Hernandez-Castro, Juan M Estevez-Tapiador, and Arturo Ribagorda. Lightweight cryptography for low-cost rfid tags. *Security in RFID and Sensor Networks*, pages 121–150, 2016.

[72] Pedro Peris-Lopez, Tong-Lee Lim, and Tieyan Li. Providing stronger authentication at a low cost to RFID tags operating under the EPCglobal framework. In *Embedded and Ubiquitous Computing, 2008. EUC'08. IEEE/IFIP International Conference on*, volume 2, pages 159–166. IEEE, 2008.

[73] Thomas Plos, Hannes Groß, and Martin Feldhofer. Implementation of symmetric algorithms on a synthesizable 8-bit microcontroller targeting passive rfid tags. In

International Workshop on Selected Areas in Cryptography, pages 114–129. Springer, 2010.

[74] Axel Poschmann, Gregor Leander, Kai Schramm, and Christof Paar. New lightweight crypto algorithms for RFID. In *Circuits and Systems, 2007. ISCAS 2007. IEEE International Symposium on*, pages 1843–1846. IEEE, 2007.

[75] Axel York Poschmann. Lightweight cryptography - cryptographic engineering for a pervasive world. *PhD thesis, Faculty of Electrical Engineering and Information Technology, Ruhr-University Bochum,Germany*, 2009.

[76] FIPS Pub. Standards for security categorization of federal information and information systems. *NIST FIPS*, 199, 2004.

[77] NIST FIPS PUB. 140-2: Security requirements for cryptographic modules. *Information Technology Laboratory, National Institute of Standards and Technology*, 2001.

[78] Quan Qian, Yan-Long Jia, and Rui Zhang. A lightweight rfid security protocol based on elliptic curve crytography. *IJ Network Security*, 18(2):354–361, 2016.

[79] Arash Reyhani-Masoleh, Mostafa Taha, and Doaa Ashmawy. New area record for the aes combined s-box/inverse s-box. In *2018 IEEE 25th Symposium on Computer Arithmetic (ARITH)*, pages 145–152. IEEE, 2018.

[80] Andrea Ricci, Matteo Grisanti, Ilaria De Munari, and Paolo Ciampolini. Design of a 2 μw rfid baseband processor featuring an aes cryptography primitive. In *2008 15th IEEE International Conference on Electronics, Circuits and Systems*, pages 376–379. IEEE, 2008.

[81] Sören Rinne, Thomas Eisenbarth, and Christof Paar. Performance analysis of contemporary light-weight block ciphers on 8-bit microcontrollers. In *Ecrypt Workshop SPEED*, pages 33–43. Citeseer, 2007.

[82] Ronald Rivest and S Dusse. The md5 message-digest algorithm, 1992.

[83] Matthew Robshaw. The eSTREAM project. *Lecture Notes in Computer Science*, 4986:1–6, 2008.

[84] Markku-Juhani O Saarinen and Daniel W Engels. A Do-It-All-Cipher for RFID: Design Requirements. *IACR Cryptology EPrint Archive*, 2012:317, 2012.

[85] Alanson P Sample, Daniel J Yeager, Pauline S Powledge, Alexander V Mamishev, and Joshua R Smith. Design of an rfid-based battery-free programmable sensing platform. *IEEE transactions on instrumentation and measurement*, 57(11):2608–2615, 2008.

[86] Alanson P Sample, Daniel J Yeager, Pauline S Powledge, and Joshua R Smith. Design of a passively-powered, programmable sensing platform for UHF RFID systems. In *RFID, 2007. IEEE International Conference on*, pages 149–156. IEEE, 2007.

[87] Taizo Shirai, Kyoji Shibutani, Toru Akishita, Shiho Moriai, and Tetsu Iwata. The 128-bit blockcipher CLEFIA. In *FSE*, volume 4593, pages 181–195. Springer, 2007.

[88] Secure Hash Standard. Fips pub 180-1. *National Institute of Standards and Technology*, 17:15, 1995.

[89] Hung-Min Sun and Wei-Chih Ting. A gen2-based RFID authentication protocol for security and privacy. *IEEE Transactions on Mobile Computing*, 8(8):1052–1062, 2009.

[90] Pim Tuyls and Lejla Batina. RFID-tags for anti-counterfeiting. In *Cryptographers' Track at the RSA Conference*, pages 115–131. Springer, 2006.

[91] Gangqiang Yang, Bo Zhu, Valentin Suder, Mark D Aagaard, and Guang Gong. The simeck family of lightweight block ciphers. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 307–329. Springer, 2015.

[92] Daniel J Yeager, Alanson P Sample, Joshua R Smith, and Joshua R Smith. Wisp: A passively powered uhf rfid tag with sensing and computation. *RFID handbook: Applications, technology, security, and privacy*, pages 261–278, 2008.

[93] Jenny Yu. Area and energy optimizations in asic implementations of aes and present block ciphers. Master's thesis, University of Waterloo, 2020.

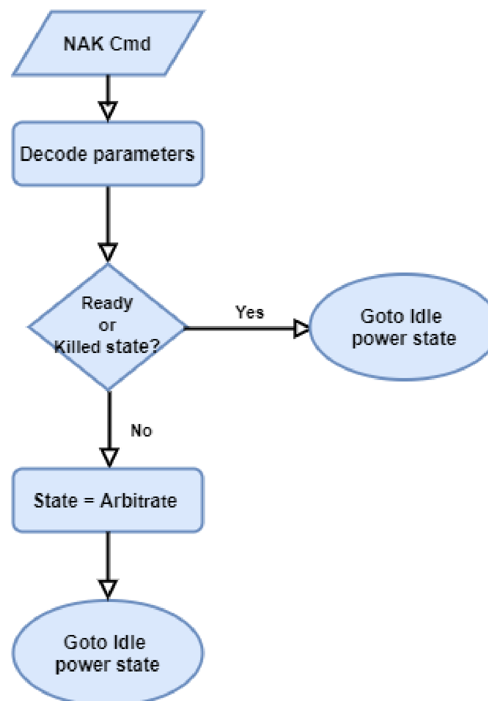
[94] Jenny W. Yu and Mark D. Aagaard. Benchmarking and optimizing AES for lightweight cryptography on ASICs. *NIST Workshop on Lightweight Cryptography*, 2019.

APPENDICES

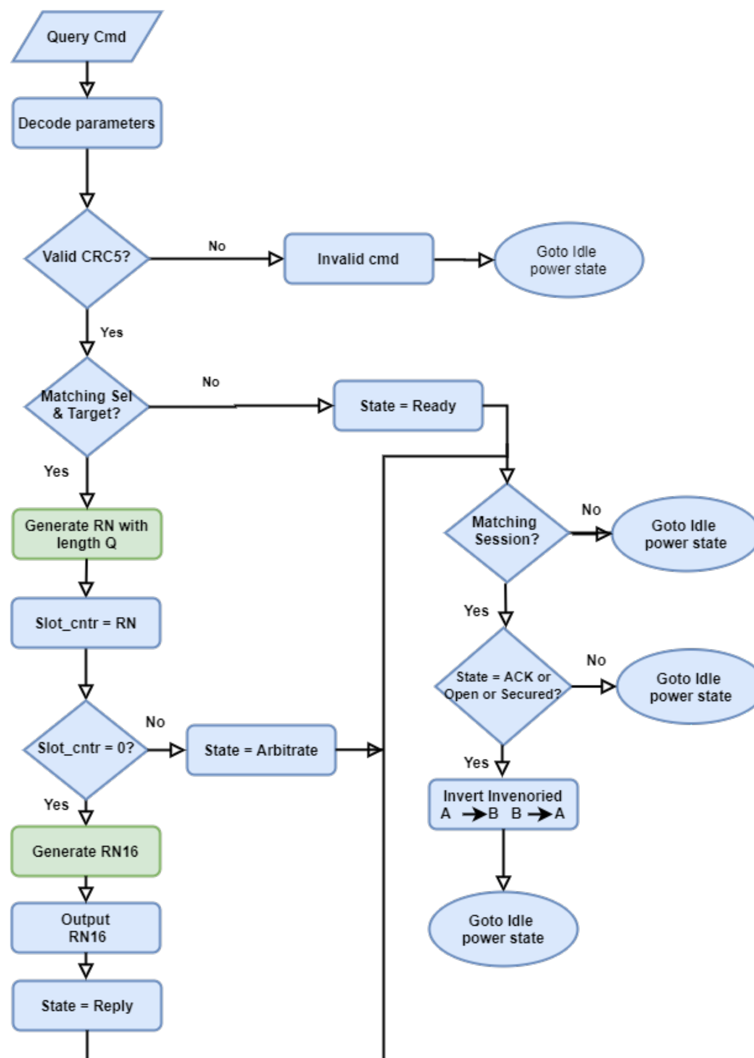
Appendix A

EPC commands flowcharts

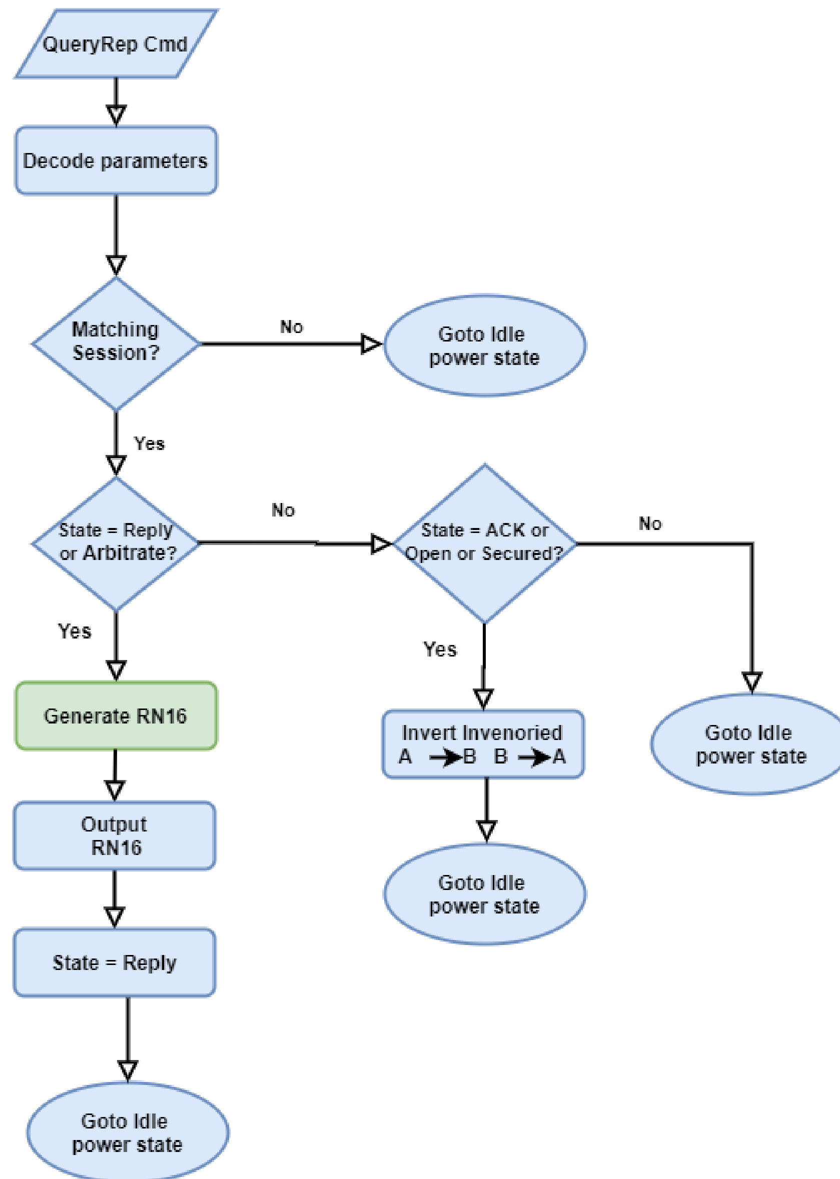
A.1 NAK command



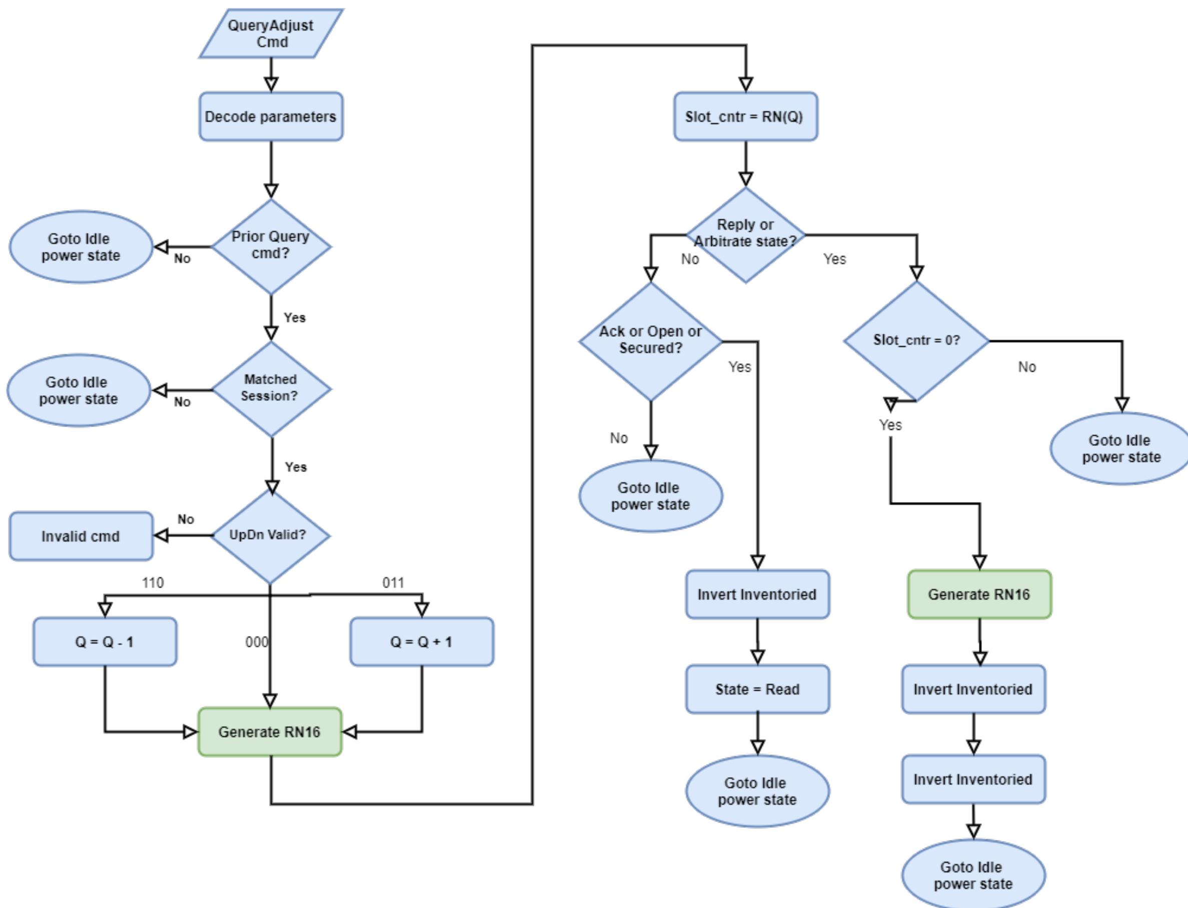
A.2 Query command



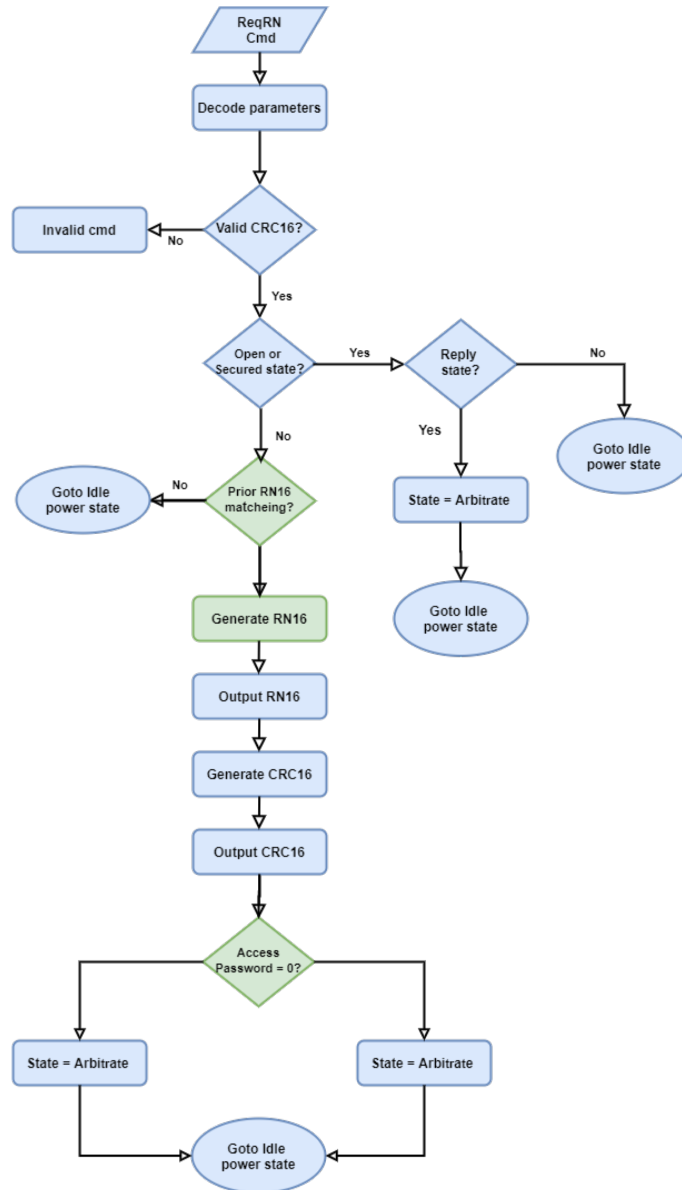
A.3 QueryRep command



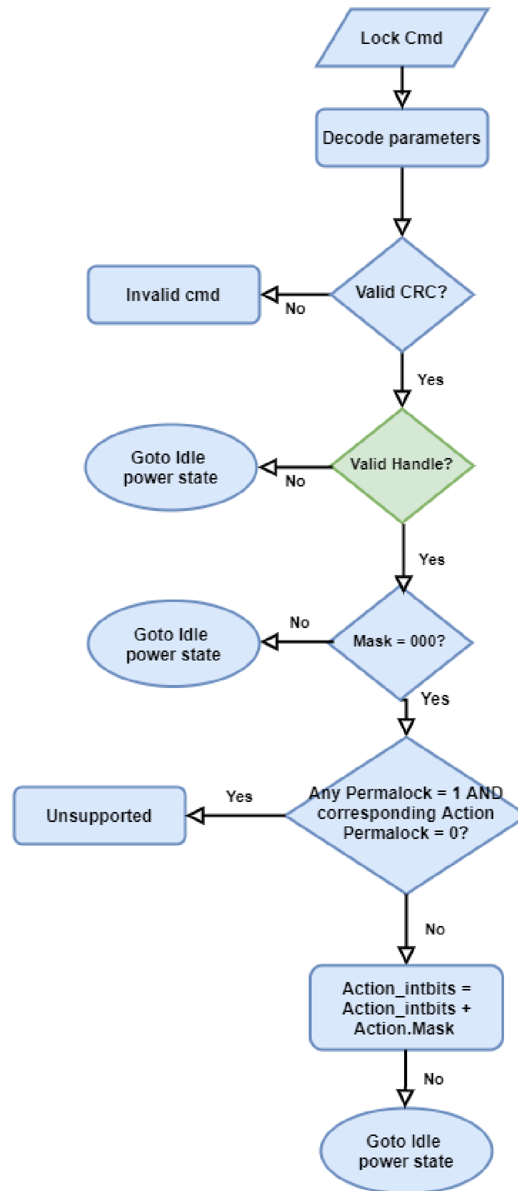
A.4 QueryAdjust command



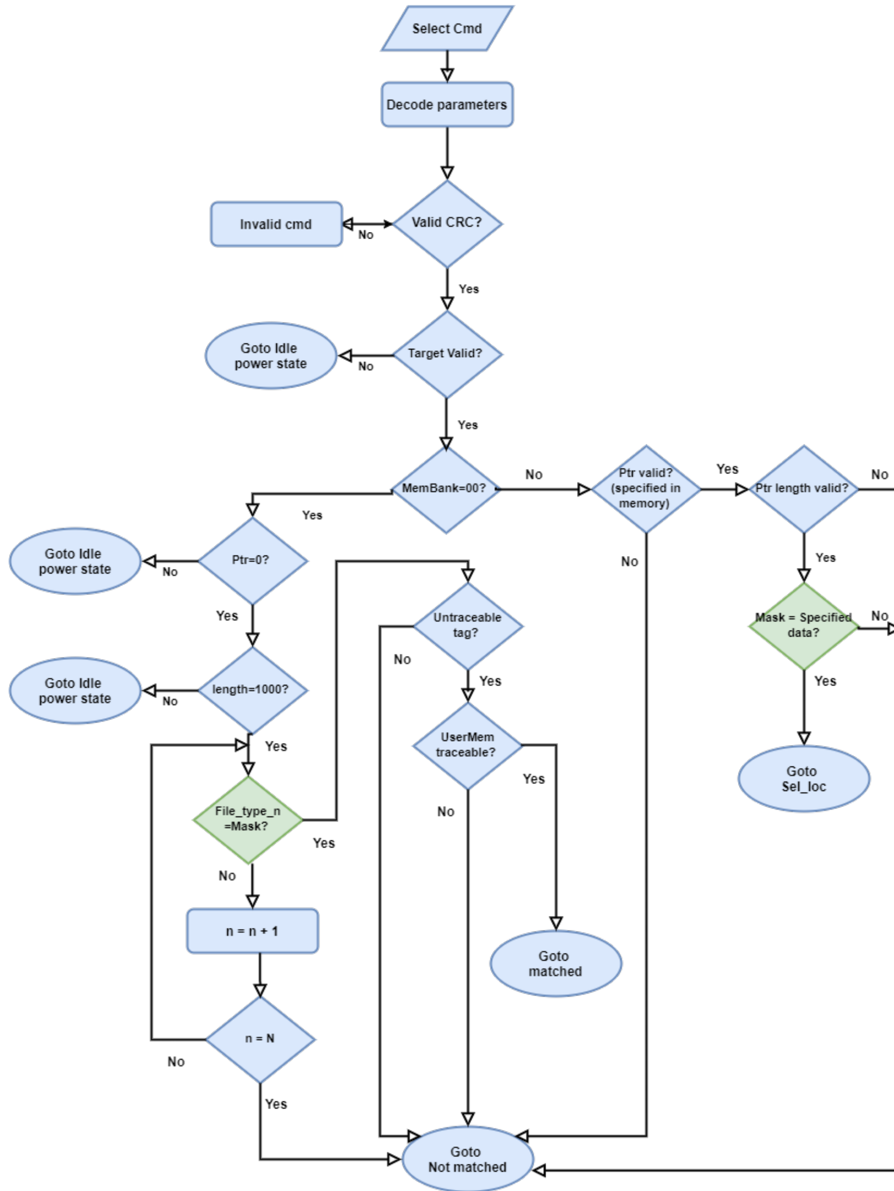
A.5 ReqRN command

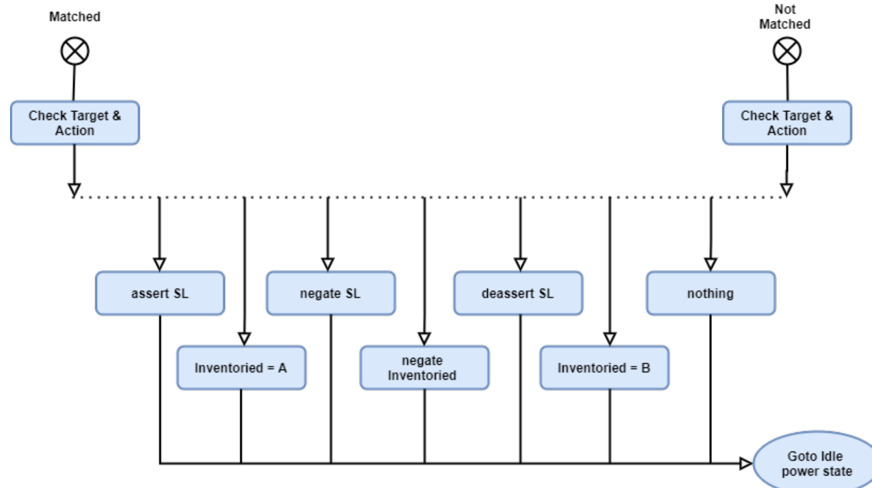


A.6 Lock command

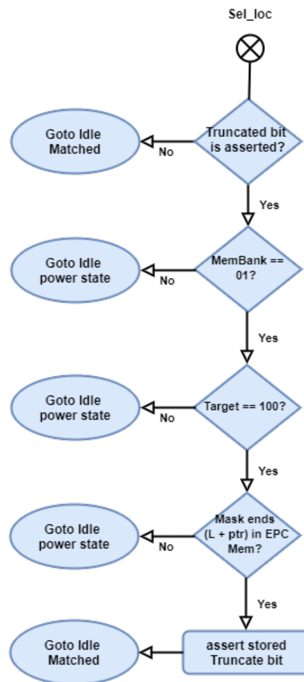


A.7 Select command



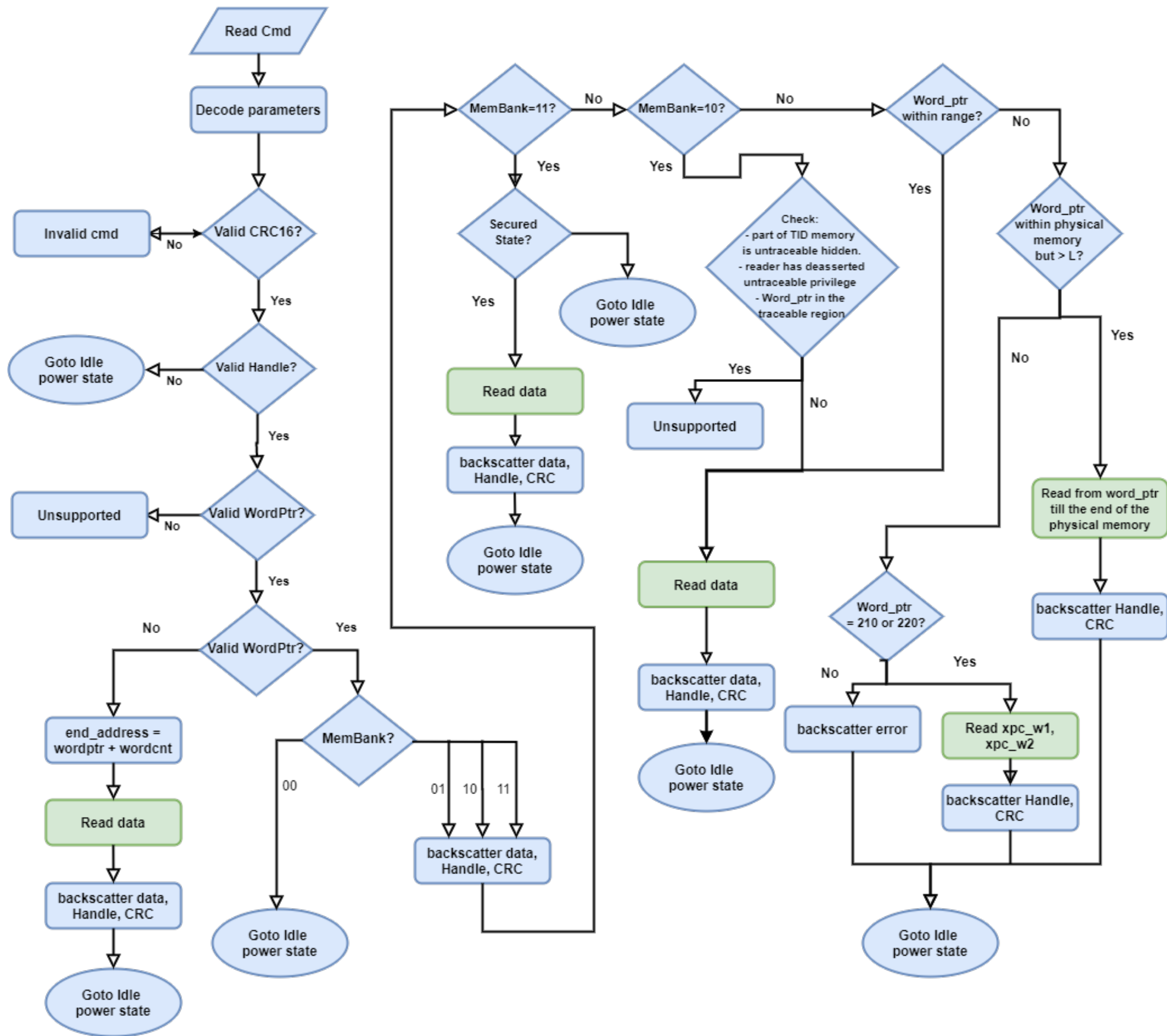


Matched
Not Matched
Check Target & Action
Check Target & Action
assert SL
negate SL
deassert SL
nothing
Inventoried = A
negate Inventoried
Inventoried = B
Goto Idle power state

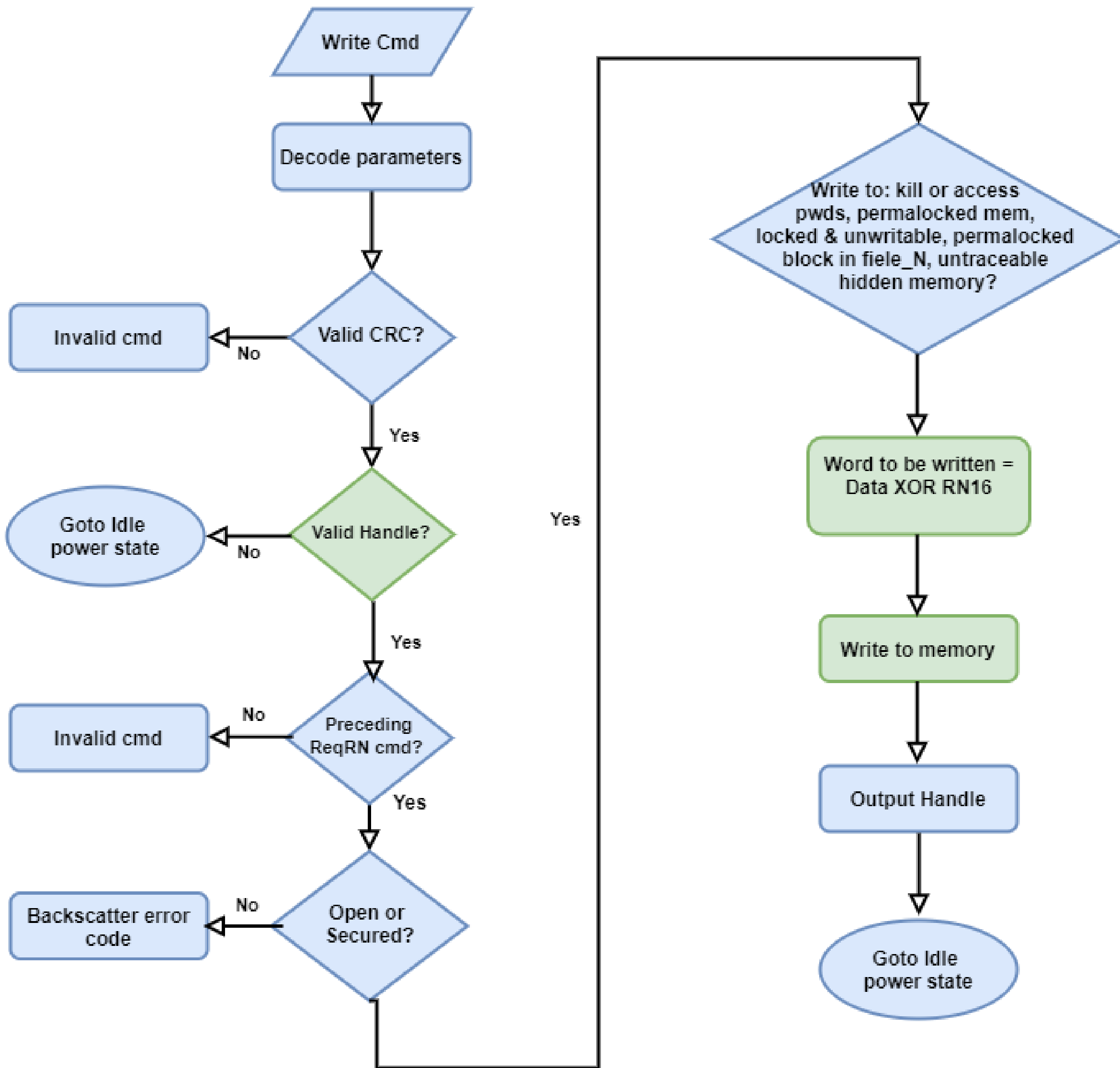


Sel_loc
Truncated bit is asserted?
No
Goto Idle Matched
Yes
MemBank == 01?
No
Goto Idle power state
Yes
Target == 100?
No
Goto Idle power state
Yes
Mask ends (L + ptr) in EPC Mem?
No
Goto Idle power state
Yes
assert stored Truncate bit
Goto Idle Matched

A.8 Read command



A.9 Write command



Appendix B

Instruction-set architecture

B.1 Memory/move instructions

Instruction	Format	Assembly	Semantics
LD	R	`ld` r_d, r_s	$r_d = Mem[r_s]$
ST	R	`st` $r_s,\ r_d$	$Mem[r_s] = r_d$
LDL	I_2	`ldl` r_d, imm_8	$r_d[7:0] = imm_8$

B.2 Arithmetic instructions

Instruction	Format	Assembly	Semantics
MOV	R	`mov` r_d, r_s	$r_d = r_s$
NOT	R	`not` r_d, r_s	$r_d = not\ r_s$
ADD	R	`add` r_d, r_d	$r_d = r_d + r_s$
SUB	R	`sub` r_d, r_d	$r_d = r_d - r_s$
AND	R	`and` r_d, r_d	$r_d = r_d\ and\ r_s$
OR	R	`or` r_d, r_d	$r_d = r_d\ or\ r_s$
XOR	R	`xor` r_d, r_d	$r_d = r_d\ xor\ r_s$
SHR	R	`shr` r_d, r_s	$r_d = r_s >> 1$
SHL	R	`shl` r_d, r_s	$r_d = r_s << 1$
SHR8	R	`shr8` r_d, r_s	$r_d = r_s >> 8$
SHL8	R	`shl8` r_d, r_s	$r_d = r_s << 8$

B.3 Branch instructions

Instruction	Format	Assembly	Semantics

BEQ	I_1	`beq` r_d	if $Zero_flag == 0$ then
			pc = pc + imm_8
			else
			pc = pc +1

BNEQ	I_1	`bneq` r_s	if $Zero_flag$!= 0 then
			pc = pc + imm_8
			else
			pc = pc +1

JMP	I_1	`jmp` r_d	pc = pc + imm_8
BSR	I_1	`bsr` r_s	Stack[*sptr*] = pc *sptr* -- pc = r_s
RTS	I_1	`rts`	*sptr* ++ pc = Stack[*sptr*]

B.4 Output instructions

Instruction	Format	Assembly	Semantics
PUT	I_1	`ldl` r_1, imm_8 `PUT` #8	$r_1[7:0] = imm_8$ $OPCNTR \leftarrow 8$ While $(OPCNTR > 0)$ $\{R_1 \leftarrow R_1 << 1$ $OPCNTR \leftarrow OPCNTR - 1\}$
Any R-type inst	R	`mov` r_1, r_s	$r_1 = r_s$

B.5 Custom instructions

Instruction option	Assembly	Semantics
CRYPTO.INIT	`CRYPTO.INIT`	crypto_module.*mode* ← #init_crypto_module crypto_module.*i_valid* ← 1 crypto_module.*data_in* ← R_s (Data on *data_in* isn't used by the cryptomodule in this option)
CRYPTO.LOAD	`CRYPTO.LOAD` R_3	crypto_module.*mode* ← #load_crypto_module crypto_module.*data_in* ← R_3 crypto_module.*i_valid* ← 1
CRYPTO.RUN	`CRYPTO.RUN` R_4, R_3	crypto_module.*mode* ← #run_crypto_module crypto_module.*data_in* ← R_3 crypto_module.*i_valid* ← 1