Implementation and Benchmarking of a Crypto Processor for a NB-IoT SoC Platform

LUIS CAVO
SÉBASTIEN FUHRMANN

MASTER´S THESIS

DEPARTMENT OF ELECTRICAL AND INFORMATION TECHNOLOGY
FACULTY OF ENGINEERING | LTH | LUND UNIVERSITY

Implementation and Benchmarking of a Crypto Processor for a NB-IoT SoC Platform

Department of Electrical and Information Technology
Lund University

MASTER OF SCIENCE THESIS

— December 10, 2017 —

Examiner:
ERIK LARSSON

Supervisors:
LIANG LIU
MICHAL STALA

Authors:
LUIS CAVO
SÉBASTIEN FUHRMANN

Abstract

The goal of this Master's Thesis is to investigate the implementation of cryptographic algorithms for IoT and how these encryption systems can be integrated in a NarrowBand IoT platform. Following 3^{rd} Generation Partnership Project (3GPP) specifications, the Evolved Packet System (EPS) Encryption Algorithms (EEA) and EPS Integrity Algorithms (EIA) have been implemented and tested. The latter are based on three different ciphering algorithms, used as keystream generators: Advanced Encryption Standard (AES), SNOW 3G and ZUC. These algorithms are used in Long Term Evolution (LTE) terminals to perform user data confidentiality and integrity protection.

In the first place, a thorough study of the algorithms has been conducted. Then, we have used Matlab to generate a reference model of the algorithms and the High-Level Synthesis (HLS) design flow to generate the Register-Transfer Level (RTL) description from algorithmic descriptions in C++. The keystream generation and integrity blocks have been tested at RTL level. The confidentiality block has been described along with the control, datapath and interface block at a RTL level using System C language. The hardware blocks have been integrated into a processor capable of performing hardware confidentiality and integrity protection: the crypto processor. This Intellectual Property (IP) has been integrated and tested in a cycle accurate virtual platform. The outcome of this Master's Thesis is a crypto processor capable of performing the proposed confidentiality and integrity algorithms under request.

Popular Science Summary

The Internet of Things (IoT) is one of the big revolutions that our society is expected to go through in the near future. This represents the inter-connection of devices, sensors, controllers, and any items, refereed as things, through a network that enables machine-to-machine communication. The number of connected devices will greatly increase. The applications taking advantage of IoT will enable to develop a great amount of technologies such as smart homes, smart cities and intelligent transportation. The possibilities allowed are huge and not yet fully explored.

Picture yourself in the near future having a nice dinner with some friends. Then, you suddenly recall that your parking ticket expires in five minutes and unfortunately your car is parked some blocks away. You are having a good time and feel lazy to walk all the way to where you parked your car to pay for a time extension. Luckily enough, the parking meter is part of the IoT network and allows you, with the recently installed new application in your smart-phone, to pay this bill from anywhere you are. This payment will be sent to the parking meter and your time will be extended. Problem solved, right?

Well, the risk comes when you perform your payment, not knowing that your "worst enemy" has interceded this communication and is able to alter your transaction. Perhaps, this individual decides to cancel your payment and you will have to pay a fine. Or even worse, this person steals your banking details and uses your money to take the vacations you've always wanted.

There are many examples in our everyday life where we expose our personal information. With an increasing number of devices existing and using wireless communications without the action of an human, the security is a key aspect of IoT. This Master's Thesis addresses the need to cover these security breaches in a world where an increasing amount of devices are communicating with each other. With the expansion of IoT where billions of devices will be connected wirelessly, our data will be widely spread over the air. The user will not be able

to protect their sensible data without these securing capabilities. Therefore, different security algorithms used in today's and tomorrow's wireless technologies have been implemented on a chip to secure the communication. The confidentiality and integrity algorithms aim to solve the two aspects of the problem: protect the secrecy of banking details and prevent the alteration of the communication's information.

In this Master's Thesis we have developed a hardware processor for securing data during a wireless communication, specifically designed for IoT applications. The developed system is realized with minimal area and power in mind, so that they can be fitted even in the smallest devices. We have compared many different hardware architectures, and after exploring many possible implementations, we have implemented the security algorithms on a hardware platform.

We believe the content of this Thesis work is of great interest to anybody interested in hardware security applied to the IoT field. Furthermore, due to the processes and methodology used in this work, it will also be of interest to people who want to know more about how higher level programming languages can be used to describe such a specialized circuit, like one performing security algorithms. Finally, people interested in hardware and software co-simulation will find in this project a good example of the utilization of such system modeling technique.

Acknowledgements

First of all, we would like to thank Michal Stala and Magnus Midholt for their trust in having offered us this great Master's Thesis opportunity, first as part of Mistbase, then at ARM. We also show our gratitude towards all our colleagues at ARM who supported us and participated to create the pleasant environment during our time at ARM.

We are grateful to Liang Liu for accepting to supervise our work on this thesis and providing us with excellent ideas and support. Furthermore we thank Erik Larsson, our examiner, for the helpful feedback on the report he provided us with. We would like to express our thank to all our teachers and teachers' assistants at EIT who shared their knowledge during these 2 years.

Last but not least, we would like to thank our families and partners for their love and support that carried us through all the challenges we have faced.

Luis Cavo & Sébastien Fuhrmann

Table of Contents

List of Figures

List of Tables

List of Algorithms

Acronyms

3G	Third Generation.
3GPP	3^{rd} Generation Partnership Project.
AC	Algorithmic C.
AES	Advanced Encryption Standard.
AHB	AMBA High-performance Bus.
AMBA	Advanced Microcontroller Bus Architecture.
ARIB	Association of Radio Industries and Businesses.
ASIC	Application-Specific Integrated Circuit.
ASMD	Algorithmic State Machine with Datapath.
ATIS	Alliance for Telecommunications Industry Solutions.
CCSA	China Communications Standards Association.
CMAC	Cipher-based MAC.
CPU	Central Processing Unit.
CTR	Counter.
DACAS	Data Assurance and Communication Security Research Center of the Chinese Academy of Sciences.
EEA	EPS Encryption Algorithm.
EIA	EPS Integrity Algorithm.
eMTC	enhanced MTC.
EPS	Evolved Packet System.
ETSI	European Telecommunications Standards Institute.
FIFO	First-In, First-Out.
FPGA	Field-Programmable Gate Array.

FSM	Finite-State Machine.
GF	Galois Field.
GPRS	General Packet Radio Service.
GSM	Global System for Mobile Communications.
HLS	High-Level Synthesis.
IF	Interface.
IoT	Internet of Things.
IP	Intellectual Property.
IV	Initialization Variable.
kGE	kilo-Gate Equivalent.
LFSR	Linear-Feedback Shift Register.
LSB	Least Significant Bit.
LTE	Long-Term Evolution.
LUT	LookUp Table.
MAC	Message Authentification Code.
MSB	Most Significant Bit.
MTC	Machine-Type Communications.
NB-IoT	NarrowBand IoT.
NIST	National Institute of Standards and Technology of the United States.
QoS	Quality of Service.
RAM	Read-Access Memory.
RO	Read-Only.
ROM	Read-Only Memory.
RTL	Register-Transfer Level.
RW	Read-Write.
SAE	System Architecture Evolution.
SoC	System on Chip.
TR	Technical Report.
TS	Technical Specification.
TSDSI	Telecommunications Standards Development Society, India.
TTA	Telecommunications Technology Association.
TTC	Telecommunication Technology Committee.
UEA	UMTS Encryption Algorithm.

UIA	UMTS Integrity Algorithm.
UMTS	Universal Mobile Telecommunications System.
UUT	Unit Under Test.
WO	Write-Only.

Chapter 1

Background

In this chapter, we will first describe the context in which this Master's Thesis is encompassed. The importance of security during a communication will be addressed, discussing the main problems and the reasons leading to our solution. Furthermore NarrowBand IoT (NB-IoT), the recently introduced radio interface by 3rd Generation Partnership Project (3GPP) will be briefly introduced, focusing on the security architecture defined for this and other Long-Term Evolution (LTE) technologies.
In this context, we will detail the specifications followed to implement the security algorithms for NB-IoT. Then, we will briefly define some mathematical tools used in this document.
A short introduction to the High-Level Synthesis (HLS) design flow will be given. The HLS flow has been used in this Master's Thesis to generate the Register-Transfer Level (RTL) description of our hardware platform. Finally, the contributions of this thesis to state-of-the-art hardware security implementations will be detailed.

1.1 Goals and Challenges

One of the problems that arise during a communication is how to secure sensible data. The wireless communication medium is susceptible to security issues, and therefore sophisticated security architectures are required to protect the confidentiality of data during a communication.
Within the framework of Internet of Things (IoT), these cryptographic algorithms must be adapted to the needs of embedded devices with limited resources, and therefore must meet limited area and power demands. The implementation of these algorithms can be done at both hardware and software level. The latter allows the reuse of resources and enables support to different cellular standards. However, the results obtained from software implementations are found to be

slower and less secure than their hardware counterpart [28]. Furthermore, the processor subsystem in an NB-IoT node would become too overloaded with security processing if these were to be implemented as software functions.
Therefore, the security functionalities that must be implemented in this technology will be handed to a hardware accelerator capable of performing the security functions needed to secure data. The implementation of such hardware accelerator is the main objective of this Master's Thesis work. This hardware module will handle the following security algorithms:

EEA1 – EIA1	SNOW 3G based algorithms
EEA2 – EIA2	AES based algorithms
EEA3 – EIA3	ZUC based algorithms

The meaning of these codes will be explained in the following sections. For now, let us state that they represent certain algorithms that must be implemented during communication as specified in [6]. This document also specifies that the first two set of algorithms in the above list must be implemented on a NB-IoT device, whereas the last one may be implemented.
Six different processing cores, one for each EEA and EIA algorithm, can be independently developed to perform the security functions defined above. However, an unified solution in which the algorithms can share the maximum amount of hardware resources, including a common datapath and common control logic has been identified as the best solution in order to fulfill the low power, low area requirements that have been imposed for this work. This unified solution is what we call the Crypto Processor and represents the second main objective of this thesis work.

Figure 1.1 shows the different components that are going to be analyzed in this work. The challenge of this work is ambitious, since we must be able to develop the three blocks observed in this diagram, ensure they are fully compliant with 3GPP's specifications, and be able to integrate them in a common cryptographic processor.

The confidentiality block will ensure the data is protected against unwanted listeners during a communication. For this reason, the sender encrypts data with secret key and thus only the receiver, which is in possession of this secret key will be able to decrypt the message.
On the other hand, the integrity block will be used to verify that a received message has not been altered during the transmission. Both of these blocks will use a cipher block or keystream generator block to be able to effectively perform the confidentiality and integrity algorithms. The confidentiality block will use the keystream provided by the cipher block to mask the input plaintext data, hence generating secured ciphertext data. The integrity block will use the keystream provided by the cipher block to produce a Message Authentification Code (MAC) used to distinguish whether the received data is legit or not.

All the blocks previously defined will ensure the communication in the NB-IoT

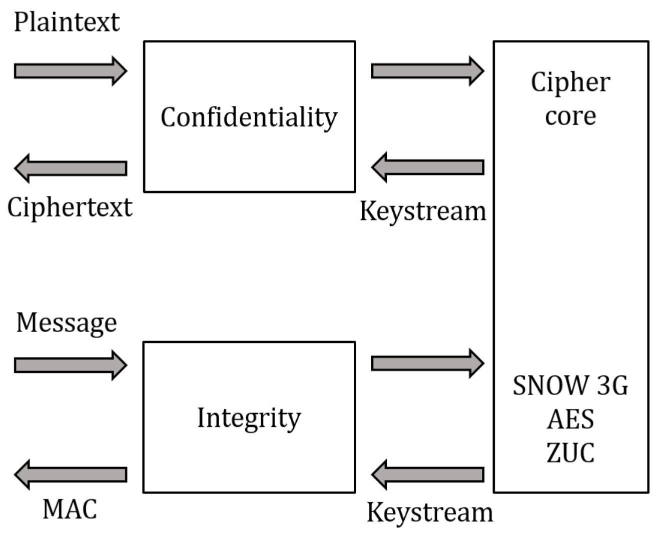


FIGURE 1.1: Simplified block diagram of the security architecture developed

node is secure. The following section will expand more on the IoT and NB-IoT, as they represent the target platform for the work developed in this Thesis work.

1.2 The Internet of Things

The IoT is enabling the deployment of a massive amount of devices which are interconnected to communicate and exchange data. It is projected that around 22 billion connected devices by 2022, of which 18 billion will be related to IoT falling in both wide-area and short-range categories [17].

Numerous technologies are being standardized to address the needs of different use cases and connectivity requirements. Successive releases of LTE have optimized Machine-Type Communications (MTC) with improved support for low-power wide-area connectivity. In LTE Release 12, low cost devices with material cost similar to General Packet Radio Service (GPRS) was introduced. In LTE Release 13, enhanced MTC (eMTC) and NB-IoT have been introduced, which provide further improvements such as device cost and complexity reduction, extended battery lifetime and enhanced coverage [37] [39] [38].

1.2.1 NarrowBand IoT

LTE NB-IoT is targeted for low-power, low-throughput applications that require excellent coverage and deployment flexibility. It is based on LTE and is therefore compatible with existing LTE networks, spectrum, and supports most LTE services, including simplifications and optimizations for low-cost, low-power and low data rate. NB-IoT standard also provides end-to-end security, which entails trusted security and authentication features.

1.2.2 3GPP Specifications

The radio interface described in the previous section, NB-IoT, was specified by 3GPP as part of Release 13. The following sections explain how these specifications are developed and maintained by 3GPP. Also, the specifications regarding the security architecture and standards used in this work will be detailed.

3GPP

The 3GPP is a collaborative group between seven telecommunication standard development organizations present around the world: Association of Radio Industries and Businesses (ARIB), Alliance for Telecommunications Industry Solutions (ATIS), China Communications Standards Association (CCSA), European Telecommunications Standards Institute (ETSI), Telecommunications Standards Development Society, India (TSDSI), Telecommunications Technology Association (TTA) and Telecommunication Technology Committee (TTC). They are referred as Organizational Partners of the 3GPP. The group was founded in 1998 with the scope of making the Third Generation (3G) derived from evolved Global System for Mobile Communications (GSM) standard globally applicable. Later, the goal of the organization with its partners widened to encompass the maintenance and development of GSM, Universal Mobile Telecommunications System (UMTS) and LTE. The 3GPP covers all aspect of the cellular communications including radio access, core transport network and the services (codecs, security, Quality of Service (QoS)).

Specifications

The specifications are defined and written by 3GPP. It is made publicly and freely available on the 3GPP specification website [3]. The documents are organized in different series of Technical Specification (TS) that cover every aspect of the communication standard. The specifications that are of interest for the security features can be found in the series 33 and 35.

Series 33 contains the documents that describes all security aspects of the standard. Technical Specification 3GPP System Architecture Evolution (SAE); Security architecture [6], numbered 33.401, includes a section dealing with the algorithms used for confidentiality and integrity protection as well as a set of test-cases with data. As we remarked in Section 1.1, Confidentiality algorithms define how the data, called plain text, is secured by encrypted it into a ciphered text. Integrity algorithms define how the data is authenticated as original unaltered information. More details are provided in Section 1.3.4 and Section 1.3.5.

Each Evolved Packet System (EPS) Encryption Algorithms (EEAs) possesses an identifier:

EIA0	Null confidentiality algorithm
128-EEA1	Confidentiality algorithm based on SNOW 3G
128-EEA2	Confidentiality algorithm based on Advanced Encryption Standard (AES)
128-EEA3	Confidentiality algorithm based on ZUC

The same exists for each EPS Integrity Algorithms (EIAs):

EIA0	Null integrity algorithm
128-EIA1	Integrity algorithm based on SNOW 3G
128-EIA2	Integrity algorithm based on AES
128-EIA3	Integrity algorithm based on ZUC

Details of the ciphers are present in other TS referenced by this main document. Excepted for AES algorithm, they are part of the series 35.

Series 35 comprises the documents related to the security algorithms. Most of them are place holders that refer to external specification resources from the owners of the algorithms. Table 1.1 list the series 35 TS.

These documents point to the specifications from ETSI and are listed in Table 1.2. EEA1 and EIA1 retake respectively the UMTS Encryption Algorithm (UEA) and the UMTS Integrity Algorithm (UIA) from ETSI. UEA2 and UIA2 are based on the stream cipher SNOW 3G.

The main Technical Specification, TS 33.401, directly refers to AES specification documents from National Institute of Standards and Technology of the United States (NIST). Table 1.3 lists these three publications. The first document describes the block cipher AES for encryption and decryption. The second document defines different modes of operation for how the cipher AES can be chained through multi call to secure a data stream. The third and last document presents the specific mode of operation used for authentication.

Data Assurance and Communication Security Research Center of the Chinese

TABLE 1.1: 3GPP Technical Specification Series 35

Number	Document title
TS 35.21x	Specification of the 3GPP Confidentiality and Integrity Algorithms UEA2 & UIA2
TS 35.215	Document 1: UEA2 and UIA2 specifications [7]
TS 35.216	Document 2: SNOW 3G specification [8]
TS 35.217	Document 3: Implementors' test data [9]
TS 35.218	Document 4: Design conformance test data [10]
TR 35.919	Document 5: Design and evaluation report [5]
TS 35.22x	Specification of the 3GPP Confidentiality and Integrity Algorithms EEA3 & EIA3
TS 35.221	Document 1: EEA3 and EIA3 specifications [11]
TS 35.222	Document 2: ZUC specification [12]
TS 35.223	Document 3: Implementors' test data [13]
TR 35.924	Document 4: Design and Evaluation Report [4]

TABLE 1.2: SNOW 3G Technical Specification and Technical Report

Place holder	Document title
TS 35.215	uea2uia2d1v21 [19]
TS 35.216	snow3gspec [20]
TS 35.217	Doc3-UEA2-UIA2-Spec-Implementors-Test-Data [23]
TS 35.218	conformance [25]
TR 35.919	uea2designevaluation [26]

TABLE 1.3: AES Technical Specification

Document title
Advanced Encryption Standard (AES) (FIPS PUB 197) [33]
Special Publication 800-38A: "Recommendation for Block Cipher Modes of Operation" [34]
Special Publication 800-38B: "Recommendation for Block Cipher Modes of Operation: The CMAC Mode for Authentication [35]

Academy of Sciences (DACAS) is the holder of the patents for the ZUC algorithms. Nevertheless these specifications are available from ETSI and are listed in Table 1.4.

TABLE 1.4: ZUC Technical Specification and Technical Report

Place holder	Document title
TS 35.221	EEA3_EIA3_specification_v1_7 [18]
TS 35.222	eea3eia3zucv16 [21]
TS 35.223	eea3eia3testdatav11 [22]
TR 35.924	EEA3_EIA3_Design_Evaluation_v2_0 [24]

1.3 Security Algorithms for NB-IoT

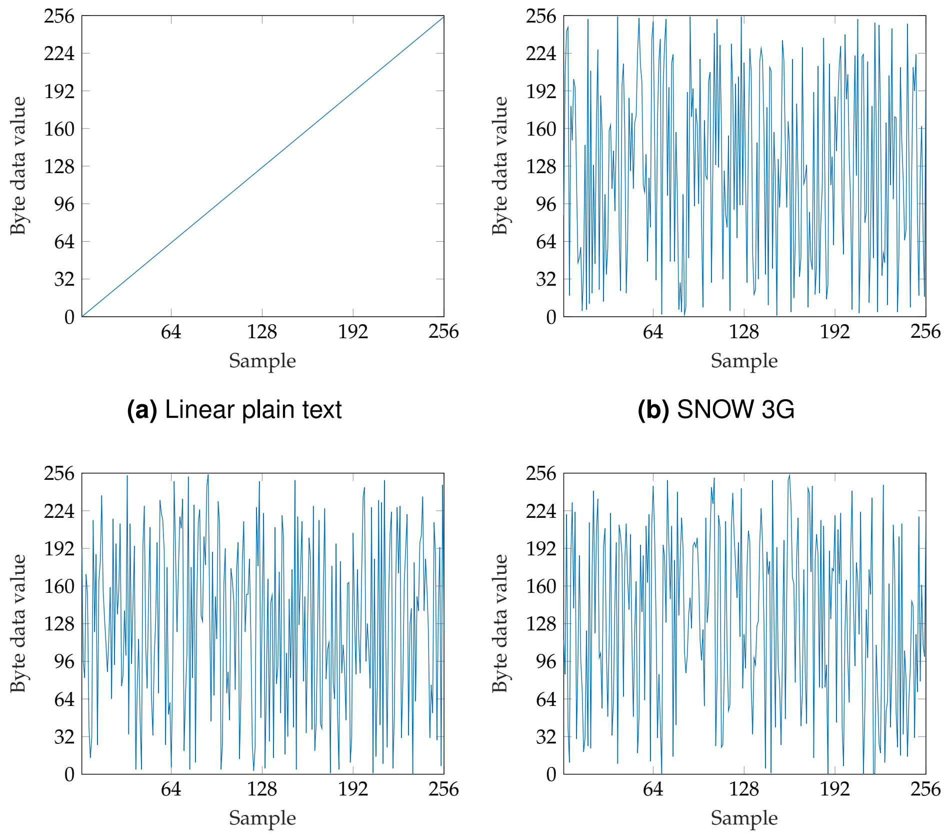
This section describes how the security algorithms defined in the security architecture are used to protect data in the NB-IoT communication standard. First, the base ciphering algorithms and their underlying concepts are introduced, followed by details on their adoption as core elements to realize the confidentiality and integrity algorithms implemented in the NB-IoT device.

1.3.1 Cipher Algorithms

The ciphering algorithms to be implemented have been defined in the previous section. AES is a block ciphering algorithm that works with fixed 4×4 bytes array. Several transformations are performed over the plain text message in order to produce the ciphered text output. The other two ciphering algorithms fall under the stream cipher algorithms type, which means that they will work on individual words. The Figure 1.2a illustrates a plain text in the form of bytes incrementing linearly. The Figure 1.2b, Figure 1.2c and Figure 1.2d show the ciphered text produced by respectively SNOW 3G, AES and ZUC algorithms. We observe how the output sequence produced is a pseudo random signal.

1.3.2 Finite Field Arithmetic

Apart from the field of mathematics, finite field are extensively used in coding theory and cryptography. Finite field arithmetic, also named Galois Field (GF) is a field containing a limited number of elements. The notation $GF(p^n)$ defines a field of p^n elements where p is a prime number called the characteristic of the field, and n a positive integer number called dimension of the field. The operations of addition, subtraction, multiplication and division are defined in this field. They are performed followed a modulo of an irreducible polynomial R. For finite field with a characteristic of 2, the addition and subtraction operations realized in the finite field become XOR.



(a) Linear plain text **(b)** SNOW 3G

(c) AES **(d)** ZUC

FIGURE 1.2: Ciphered text produced by the EEA/EIA keystream generators

Additions and subtractions are the same operation.

$$a + b \equiv a - b \equiv a \oplus b$$

The multiplication in GF is an usual multiplication with the product modulo the irreducible polynomial R that defines the finite field.

$$a \bullet b \equiv (a \times b) \bmod R$$

The division in the finite field is the multiplication by the inverse modulo the irreducible polynomial. The multiplicative inverse is obtained with the extended euclidean algorithm.

1.3.3 Substitution Boxes

Substitution Boxes, also names S-boxes, is a critical element of the security algorithms. It is used to substitutes a value by another one in order to mask the relation between the ciphered text and the key. The relation between the input and output of the substitution box must diverge as much as possible from a linear or affine function. The S-boxes defined in the NB-IoT algorithms are constant and designed to offer an increased resistance against attacks.

1.3.4 Confidentiality Algorithms

The scope of the confidentiality algorithms is to guarantee the inaccessibility to the information from unauthorized entities, keeping the information secret during a communication. The data is obscured in a way that only the knowledge of the required context, such as the key, provides the possibility of decrypting the plain text information.

1.3.5 Integrity Algorithms

These algorithms ensure the integrity of the information from the source to the target. This allows to detect that the data has not been modified or partially destroyed during the transmission. This type of algorithms provide a MAC as a result of processing a message integrity.

1.4 HLS Design Flow

A traditional hardware design flow will in most cases start with a project specification. Often, an algorithm that is able to meet the proposed requirements and specifications is developed in languages like C, C++, Matlab, etc. This description aims to tackle the functional design with almost no knowledge on the actual hardware implementation. Therefore, this high-level behavior is usually much faster to implement than the next step in the hardware design flow: the RTL design. This level of abstraction is really dependent on the actual hardware architecture of the design, caring not only on the functional side.

Furthermore, RTL is hand-coded by the design teams and therefore any error encountered in this description, any late minute changes in the specification, a change in technology or architecture can lead to long design processes. HLS emerges as a solution to the long design times in RTL description. By using HLS the designer is capable of making the architectural choices and lets the HLS implement them. These tools provide the opportunity to implement different solutions of the same algorithm by exploring the design space looking for the implementation that best matches the specifications and providing the best trade-off between power, area and performance. Furthermore, they will enable to retarget these models to new fabrication technologies making designs more reusable.

Many leading industry companies have developed a HLS during the last decade, and the interest has arisen in the past years. Cadence Stratus, Synopsys Synphony, Xilinx Vivado HLS, Mentor Catapult HLS, all claim to deliver from 5 to 10X higher design and verification productivity than traditional RTL flows, especially in the field of communications and multimedia applications.

In [36] it is shown how high level synthesis represents a powerful tool for design space exploration. In this paper they show that HLS will need of architectural refinements, especially when dealing with memory interfacing to assist and improve the results of the HLS tool.

1.4.1 Catapult Synthesis

Catapult Synthesis is a HLS tool by Mentor that takes as input the model-based description of the design, specified in C, C++ or System C, and synthesizes it to generate an RTL description. Figure 1.3 shows Catapult HLS design flow.

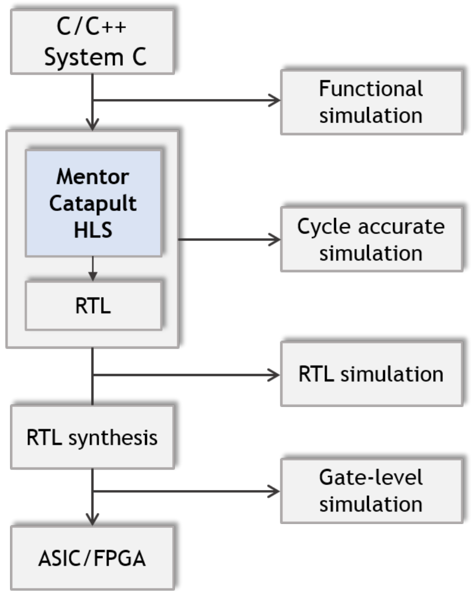


FIGURE 1.3: Catapult HLS flow

1.4.2 Frame Based vs Sample Based Processing

The input to Catapult HLS is usually an algorithmic untimed description, written in a high-level language (C or C++). Then, the designer will impose some architectural constraints to help the tool generate the desired RTL description. This is referred to as frame based processing in this document.

On the other hand sample based C++ description will describe an algorithm in a timed manner. The implementation of such algorithms will take longer time and will be less versatile but will ensure the resulting RTL is what the designer expects. In this Master's Thesis, we have used the sample based approach to describe the different hardware blocks synthesized using Catapult.

1.5 Thesis Contributions

The main contribution of this Thesis work is a cryptographic processor capable of performing the 3GPP EEA and EIA algorithms as defined in [6] for the IoT field. There are some similar works, both academic literature [15] and commercial products. Perhaps the most similar commercial processor is found in [41]. The comparison with these works will be covered in chapter 6.

This Master's Thesis uses, as described in the previous section, the HLS design flow. This, together will the hardware and software co-simulation capability provided by a virtual simulation platform has enabled for an easy architecture exploration, fast prototyping of the hardware system and agile testing and integration. No prior evidence of applying the HLS design flow to cryptography has been found.

Going into more detail into the architecture, a partial combination of two of the ciphering algorithms, SNOW 3G and ZUC, has been realized. After exploring both algorithms independently some hardware resources have been combined due to the structural similarities of these algorithms.

Also, many different computations perform in finite field arithmetic have been studied in detail, resulting in a computational model for each of these operations. For instance, the different substitution boxes (S-boxes) have been analyzed and compared against their LookUp Table (LUT) implementation. Furthermore, multiplication, division and inversion in GF has been implemented as on-the-fly computations. Finally, the Mulα and Divα operators, used in SNOW 3G, have been explored leading to a successful and optimized implementation for area minimization.

Chapter 2

Security Algorithms

The following chapter details the 128 bits ciphering algorithms used to perform the confidentiality and integrity algorithms as defined by 3GPP specifications. Furthermore, the confidentiality and integrity algorithms based on the ciphering cores, which are used as keystream generators, are also presented. Each block of the design is detailed following a top-down hierarchy, starting from the top level and going to the internal architecture of each sub-block. The hardware implementation of these blocks is presented in the following chapter.

2.1 Ciphering Algorithms

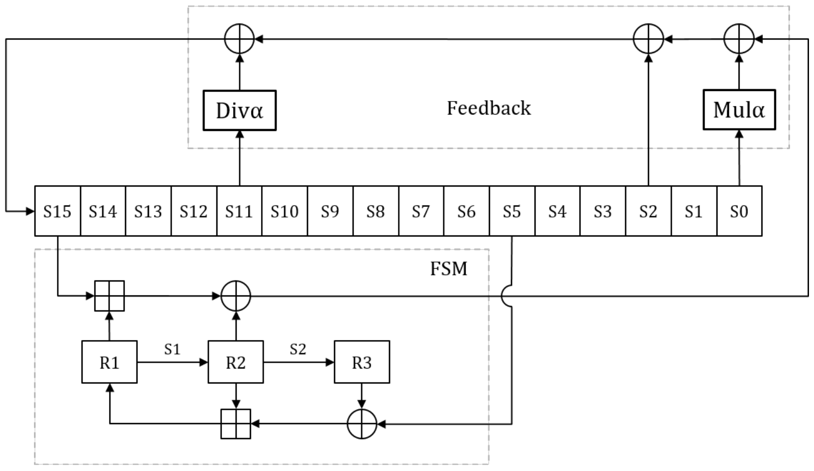
The ciphering algorithms implemented in this work are presented in the following sections. These ciphering algorithms, which are used to generate a keystream used to mask the plaintext, all use a 128-bit key. This section aims to give a theoretical view of the algorithms disregard of the hardware implementation.

2.1.1 SNOW 3G Cipher

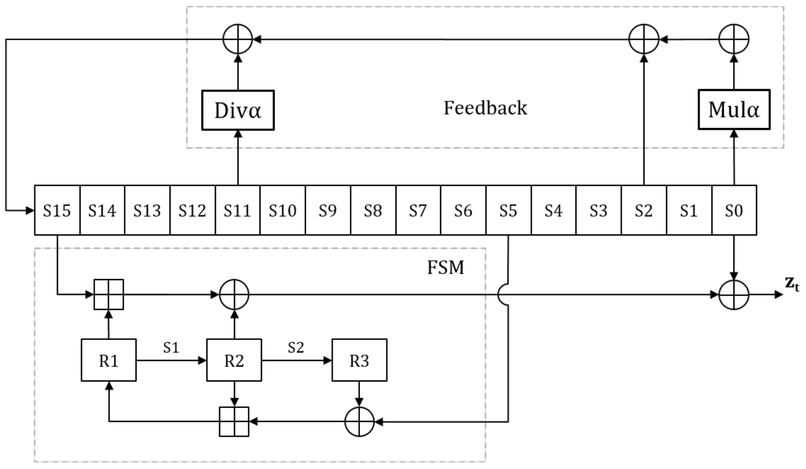
SNOW 3G is one of the three standardized algorithms in the 3GPP architecture. It is a stream cipher algorithm, which will generate a 32 bit word output, called keystream, under a 128 bit secret key and a 128 bit publicly known Initialization Variable (IV). The output word will be used to mask a plaintext input producing this way the ciphertext output.

The keystream generator is based on three major modules. The first building block is a Linear-Feedback Shift Register (LFSR) consisting of 16 registers of 32 bits. The LFSR feeds a Finite-State Machine (FSM) with its input values. The

FSM consists of three 32-bit registers, updated based on two permutation boxes and some other operators. The output of the FSM is XORed with the data from register s_0 in the LFSR or connected to the feedback loop. This third module with accept inputs from the LFSR (and the FSM out during initialization mode), and will produce an output by performing some field operations over its inputs. The main sub-blocks in the feedback path, *Mulα* and *Divα* operators, perform the aforementioned field operations.



(a) SNOW 3G during initialization mode



(b) SNOW 3G during keystream mode

FIGURE 2.1: SNOW 3G ciphering

SNOW 3G specification [20] defines two work modes, initialization mode and keystream mode. Clocking the system means shifting the LFSR to the right in

Figure 2.1 and updating the registers in the FSM. The two working modes operate as follows:

Initialization mode: during initialization mode, the LFSR is initialized using the input key and the IV. The FSM registers are initialized to zero and both the LFSR and the FSM are clocked 32 times with the output of the FSM connected to the input of the feedback path.

Keystream mode: during keystream mode, the system is clocked once discarding the first output. Then, each time the system is clocked, a 32-bit output from the FSM is produced and XORed with the content of register s_0 of the LFSR, producing an output word z_t.

The following sections give a deeper insight on the construction of SNOW 3G main building blocks.

LFSR

The LFSR consists of 16 stages of 32 bits, which results in a total of 512 bits. It is initialized during initialization mode with the key and IV as given by [20].

Feedback

The feedback polynomial is given by $f(x) = \alpha x^{16} + x^{14} + \alpha^{-1} x^{15} + 1$ defined over GF(2^{32}). SNOW 3G will use these field operators (multiplication/division) along with 32-bit XOR operations to produce its output.

The Mul$_\alpha$ and Div$_\alpha$ Operators

These functions will map an 8 bit input to a 32 bit output. It can be implemented as a precomputed table and will be used to initialize and feed the LFSR correctly. The operation of $Mul\alpha$ is governed by $Mul\alpha(x) = x^4 + \beta^{23}x^3 + \beta^{245}x^2 + \beta^{48}x + \beta^{239}$ where x takes a value between 0 and 255 and β is a root of the polynomial $x^8 + x^7 + x^5 + x^3 + 1$.

$Div\alpha$ is governed by $Div\alpha(x) = x^4 + \beta^{16}x^3 + \beta^{39}x^2 + \beta^6 x + \beta^{64}$.

Both operators can be implemented using a 1 kB table each. Different implementations will be explored in Section 3.3.

FSM

SNOW 3G FSM is composed of the following sub-blocks:

- Three 32-bit registers R1, R2 and R3.
- Two S-boxes S1 and S2, based on S-box S_R and S-box S_Q respectively.
- Two 32-bit XOR ($\oplus$).
- Two 32-bit modulo adders ($\boxplus$). The modulo adders are used as a non-linear combining function. In this manner, one modulo adder will combine s_{15} with $R1$ through $(s_{15} \boxplus R1) \oplus R2$, and the other modulo adder combines $R2$ with $(R3 \oplus s_5)$.

When clocking the FSM, the internal registers will be updated as $R3 = S_2(R2)$, $R2 = S_1(R1)$ and $R1 = (s_{15} \boxplus R1) \oplus R2$.

The details on the implementation of SNOW 3G will be given in Chapter 3, where different implementations of SNOW 3G building blocks are explored using Catapult Synthesis. In Section 4.3.1, the final architecture of SNOW 3G is explained.

2.1.2 AES Cipher

The AES is a block cipher using a fix block size of 128 bits. As defined by the specifications [33], the key size used is also 128 bits. The principle of operation of AES cipher is to pass a block of data to encrypt, defined as the state, through a network of substitutions and permutations in order to secure the original plain text. The design of this algorithm is made to allow fast processing in both software and hardware implementations [40]. The 128-bit block of data is stored as a state, which is organized in 16 bytes as represented in Figure 2.2. The block experiences several rounds of successive transformations as shown in Figure 2.3 to encrypt the plain text.

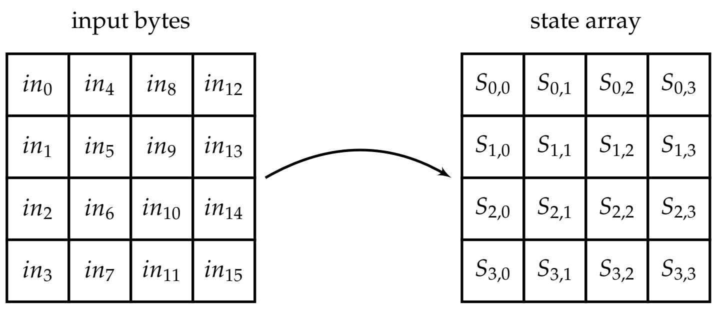

input bytes
state array
in_0 in_4 in_8 in_{12}
in_1 in_5 in_9 in_{13}
in_2 in_6 in_{10} in_{14}
in_3 in_7 in_{11} in_{15}
$S_{0,0}$ $S_{0,1}$ $S_{0,2}$ $S_{0,3}$
$S_{1,0}$ $S_{1,1}$ $S_{1,2}$ $S_{1,3}$
$S_{2,0}$ $S_{2,1}$ $S_{2,2}$ $S_{2,3}$
$S_{3,0}$ $S_{3,1}$ $S_{3,2}$ $S_{3,3}$

FIGURE 2.2: Input data block into AES state array

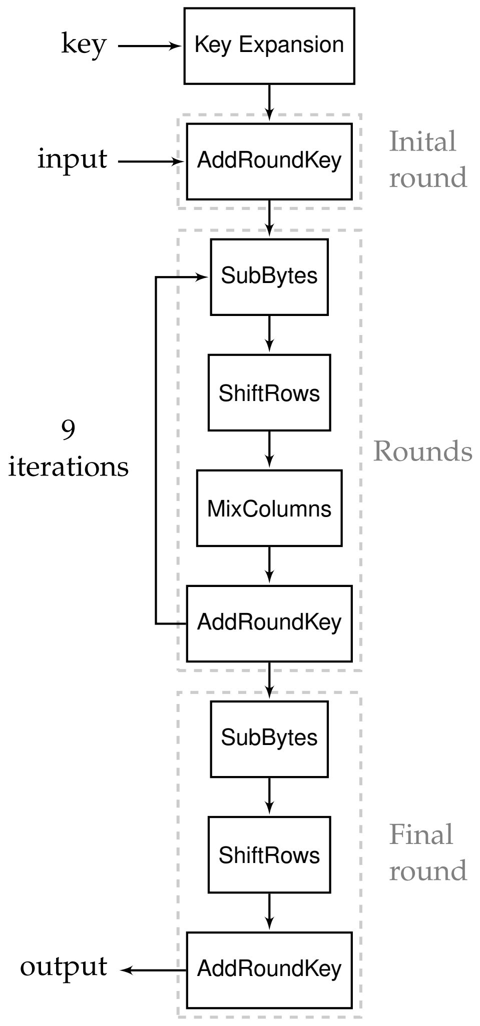


FIGURE 2.3: ASMD of AES encryption

The different stages of AES are described in Figure 2.3. The first stage is the key expansion routine, which transforms the 128 bits key into a larger expanded key resulting in a total of 44 words of 32 bits, i.e. 1408 bits. The next transformation, AddRoundKey(), consumes part of the expanded key to substitute the data state. The following four operations are recurrently called 9 times in the same order: SubBytes(), ShiftRows(), MixColumns() and AddRoundKey(). Finally, during the last round, the MixColums() transform is skipped to keep only SubBytes(), ShiftRows() and AddRoundKey(). In total, each transformation is performed 10 times, except for MixColumns() which is applied 9 times while AddRoundKey() is processed 11 times. The Key Expansion is carried out a single time. The different substitution and permutation transforms are presented with more detail later in this section.

Key Expansion

This operation grows the key in order to be used by AddRoundKey() transformation as previously specified. The 128-bit key is expanded into 44 words of 32 bits setting the total size to 1408 bits for the expanded key. The expansion routine is represented in Figure 2.4. The first 4 words are directly copied from the 4 words of the cipher key. Permutations and substitutions are then applied to the word w_3. The RotWord() operation realizes a cyclic byte shift to left. The word gets all its bytes substituted using the S-Box S_R. The word is finally XORed with a word constructed from the constant vector of bytes Rcon. Rcon is composed of 10 bytes defined by the following hexadecimal values in equation (2.1).

$$Rcon = \begin{bmatrix} 01 & 02 & 04 & 08 & 10 & 20 & 40 & 80 & 1B & 36 \end{bmatrix} \tag{2.1}$$

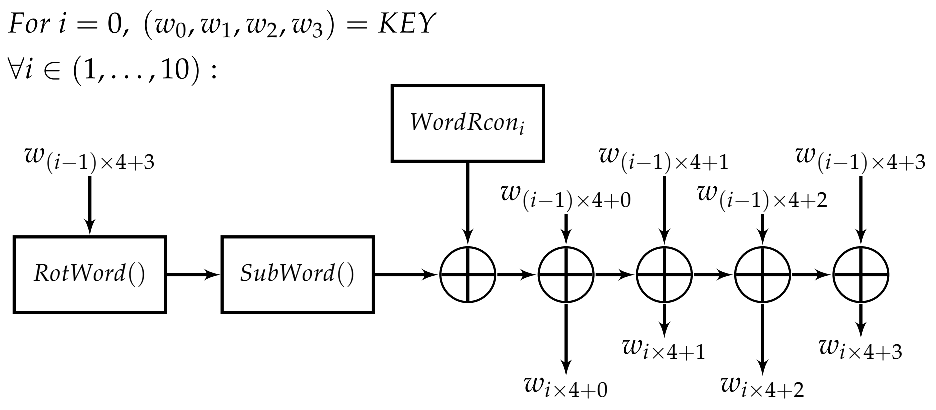


FIGURE 2.4: Key Expansion diagram

The word derived from Rcon uses one byte from the constant as its most significant byte while the other least significant bytes are set to 0. This generated word is XORed with the word from the substitution operation. The expanded key word $w_{i\times4}$ results from the XOR operation between the word $w_{(i-1)\times4}$, where i represents the iteration index, and the word resulting from the previous operation, w_{Rcon}. The next 4 words of the expanded key, $w_{i\times4+1}$, $w_{i\times4+2}$, $w_{i\times4+3}$, are constructed using the previous word $w_{i\times4+c-1}$ and the word $w_{(i-1)\times4+c}$, with c equal to 1, 2 and 3 respectively. This is summarized with the following equations:

For $c = 0$:

$$w_{i\times4+c} = w_{i\times4+(c-1)} \oplus w_{Rcon} \tag{2.2}$$

For $c = 1, 2$ and 3:

$$w_{i\times4+c} = w_{i\times4+(c-1)} \oplus w_{(i-1)\times4+c} \tag{2.3}$$

SubBytes

This substitution transform is applied on each byte of the state matrix. Using the Rijndael S-Box S_R, mapping a byte value to another byte, the data block bytes are transmuted.

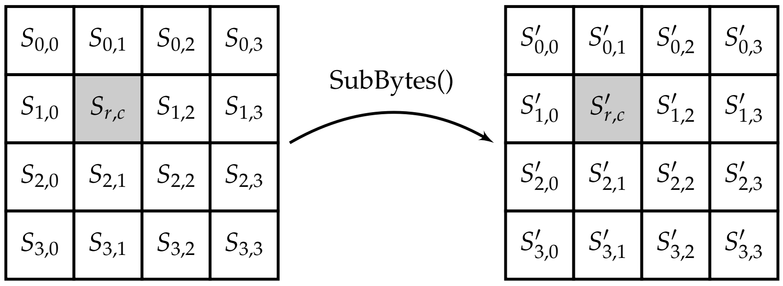


FIGURE 2.5: SubBytes() transformation

ShiftRows

This transformation permutes the bytes on the rows of the data block. It performs a cyclic shift of the bytes in each row with a different offset. The first row is unmodified. The second row gets all its bytes moved by one position to the left. The leftmost byte is moved to the rightmost place. The same operation is applied to the third and fourth row with an offset of respectively two and three. Figure 2.6 illustrates this transformation.

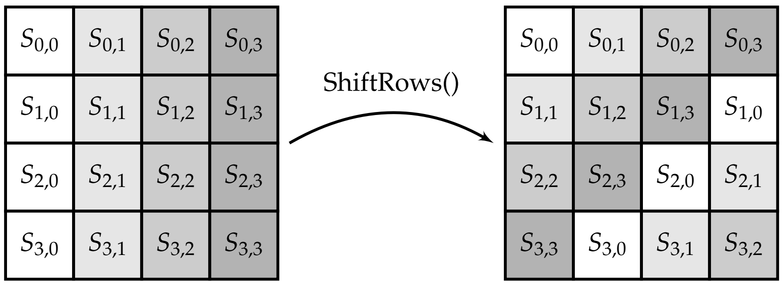


FIGURE 2.6: ShiftRows() transformation

MixColumns()

This transformation realizes the multiplication in GF(2^8) of a byte vector by a constant matrix defined in equation (2.4).

$$\begin{bmatrix} S'_{0,c} \\ S'_{1,c} \\ S'_{2,c} \\ S'_{3,c} \end{bmatrix} = \begin{bmatrix} 2 & 3 & 1 & 1 \\ 1 & 2 & 3 & 1 \\ 1 & 1 & 2 & 3 \\ 3 & 1 & 1 & 2 \end{bmatrix} \bullet \begin{bmatrix} S_{0,c} \\ S_{1,c} \\ S_{2,c} \\ S_{3,c} \end{bmatrix} \tag{2.4}$$

The resulting byte vector is replaced with the four computed bytes following equation (2.5).

$$\begin{aligned} S'_{0,c} &= 2 \bullet S_{0,c} \oplus 3 \bullet S_{1,c} \oplus S_{2,c} \oplus S_{3,c} \\ S'_{1,c} &= S_{0,c} \oplus 2 \bullet S_{1,c} \oplus 3 \bullet S_{2,c} \oplus S_{3,c} \\ S'_{2,c} &= S_{0,c} \oplus S_{1,c} \oplus 2 \bullet S_{2,c} \oplus 3 \bullet S_{3,c} \\ S'_{3,c} &= 3 \bullet S_{0,c} \oplus S_{1,c} \oplus S_{2,c} \oplus 2 \bullet S_{3,c} \end{aligned} \tag{2.5}$$

Here, $\bullet$ represents a multiplication in GF(2^8) and $\oplus$ represents an XOR, which is an addition in GF(2^8). Details about the finite field operations are given in Section 1.3.2. This multiplication operates on the state column by column, illustrated in Figure 2.7.

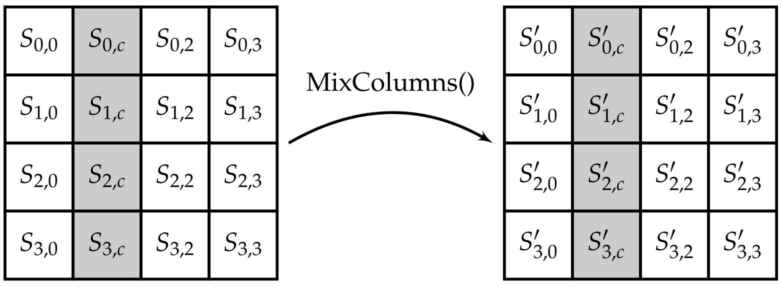

$S_{0,0}$ $S_{0,c}$ $S_{0,2}$ $S_{0,3}$
$S_{1,0}$ $S_{1,c}$ $S_{1,2}$ $S_{1,3}$
$S_{2,0}$ $S_{2,c}$ $S_{2,2}$ $S_{2,3}$
$S_{3,0}$ $S_{3,c}$ $S_{3,2}$ $S_{3,3}$
MixColumns()
$S'_{0,0}$ $S'_{0,c}$ $S'_{0,2}$ $S'_{0,3}$
$S'_{1,0}$ $S'_{1,c}$ $S'_{1,2}$ $S'_{1,3}$
$S'_{2,0}$ $S'_{2,c}$ $S'_{2,2}$ $S'_{2,3}$
$S'_{3,0}$ $S'_{3,c}$ $S'_{3,2}$ $S'_{3,3}$

FIGURE 2.7: MixColumns() transformation

AddRoundKey()

This transformation operates column wise on the state. All of the bytes in a column of the state matrix are grouped to form a 32 bits word. This word is XORed with an expanded key word. This word index is defined by $l = round * Nb$, Nb being the number of words in a state and *round* the index of the round iteration as previously defined. Each of the 44 words composing the expanded key is used. Figure 2.8 illustrates the transformation by showing the state words and the expanded key words. One AddRoundKey() transformation is composed of 4 XOR operations on 32 bits words.

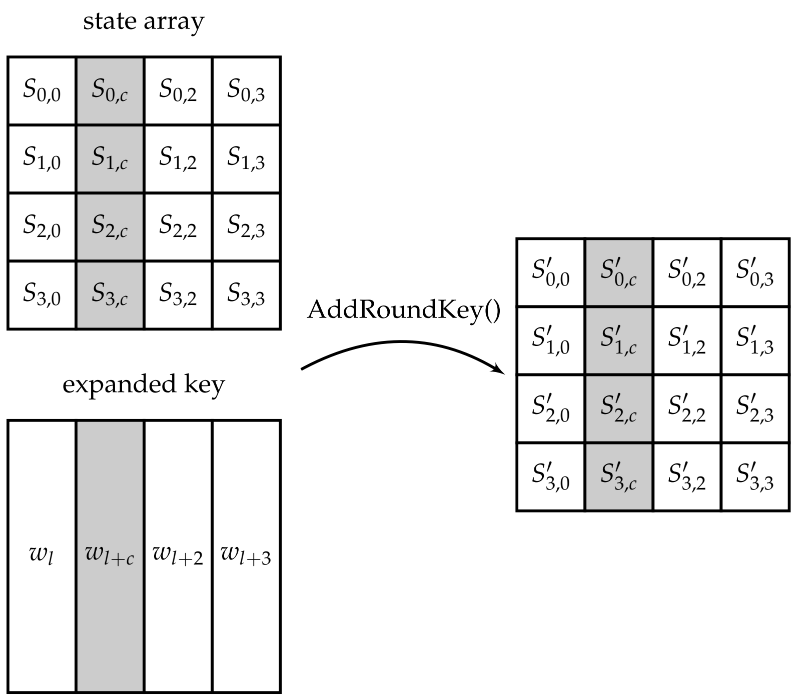

state array
$s_{0,0}$ $s_{0,c}$ $s_{0,2}$ $s_{0,3}$
$s_{1,0}$ $s_{1,c}$ $s_{1,2}$ $s_{1,3}$
$s_{2,0}$ $s_{2,c}$ $s_{2,2}$ $s_{2,3}$
$s_{3,0}$ $s_{3,c}$ $s_{3,2}$ $s_{3,3}$
AddRoundKey()
expanded key
w_l w_{l+c} w_{l+2} w_{l+3}
$s'_{0,0}$ $s'_{0,c}$ $s'_{0,2}$ $s'_{0,3}$
$s'_{1,0}$ $s'_{1,c}$ $s'_{1,2}$ $s'_{1,3}$
$s'_{2,0}$ $s'_{2,c}$ $s'_{2,2}$ $s'_{2,3}$
$s'_{3,0}$ $s'_{3,c}$ $s'_{3,2}$ $s'_{3,3}$

FIGURE 2.8: AddRoundKey() transformation

2.1.3 Inverse Cipher

Being the AES algorithm a block cipher, a different flow is defined for decryption. Most of the operations operate following the same principles defined for the encryption process.

Key Expansion remains exactly the same.

InvAddRoundKey() transformation is the inverse cipher version of the operation AddRoundKey(). The difference is the expanded key word utilization order. InvAddRoundKey() processes the expanded key word in reverse order.

InvShiftRows() applies the same transformation as ShiftRows() except for the cyclic shift being to the right instead of the left.

InvSubBytes() substitutes the bytes of the state as SubBytes() transformation but with a different S-Box called Inverse S_R.

InvMixColumns() transformation that performs the multiplication in GF(2^8) of a byte vector by a constant 4×4 matrix. The operation is as specified by MixColumns() except for the constant polynomial matrix, which is defined

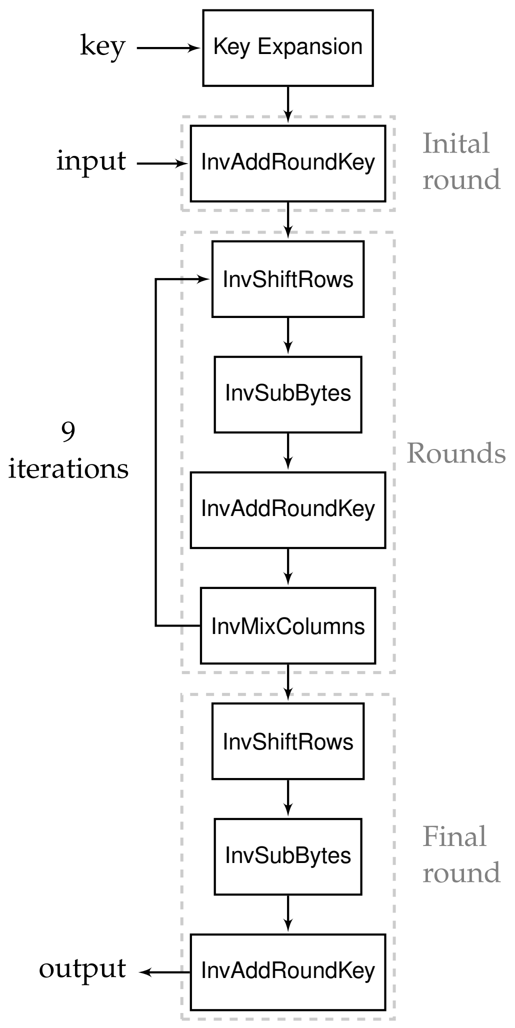


FIGURE 2.9: ASMD of AES decryption

by the hexadecimal values in equation (2.6).

$$M_{poly\ inv} = \begin{bmatrix} 0E & 0B & 0D & 09 \\ 09 & 0E & 0B & 0D \\ 0D & 09 & 0E & 0B \\ 0B & 0D & 09 & 0E \end{bmatrix} \tag{2.6}$$

The AES inverse cipher processes the different transformations in a different order. Figure 2.9 illustrates the flow. During the rounds, the order of InvShiftRows() and InvSubBytes() are inversed compare the the transformation ShiftRows() and SubBytes() in the AES flow for encryption. The same rearrangement for InvAddRoundKey() and InvMixColumns() can be noticed.

2.1.4 ZUC Cipher

ZUC is a word-oriented stream cipher algorithm, which takes a 128 bit secret key and a 128 bit publicly known IV as input and generates a 32 bit word output, called keystream. The output word will be used to mask a plaintext input producing this way the ciphertext output.

The keystream generator is based on four major modules: a LFSR, a feedback path which we separate from the LFSR module for convenience, a non-linear FSM, and bit reorganization module. The LFSR consists of 16 registers of 31 bits each and the feedback path is constructed by a primitive polynomial in $\mathrm{GF}(2^{31}-1)$. This is a major difference with SNOW 3G algorithm, since ZUC will produce sequences over the prime field $\mathrm{GF}(2^{31}-1)$ instead of being over a $\mathrm{GF}(2^{m})$, like SNOW 3G cipher. This methodology for generating sequences contributes to this algorithm's resistance to bit-oriented cryptographic attacks, fast correlation attacks, linear distinguishing attacks and algebraic attacks [4].

The LFSR feeds a FSM with its input values. The bit reorganization module will take the higher or lower 16 bits of some LFSR registers as shown in Figure 2.10, and will create 32-bit words that become inputs to the FSM. The FSM consists of two 32-bit registers, updated based on two linear functions *L1* and *L2* and two non-linear transformations S0 and S1.

The output W of the FSM is XORed with the data from X_3, which is formed from the concatenation of the lower 16 bits of s_2 and the higher 16 bits of s_0, therefore $X_3 = s_{2L}\|s_{0H}$. The feedback has 6 inputs and are formed as depicted in Figure 2.10.

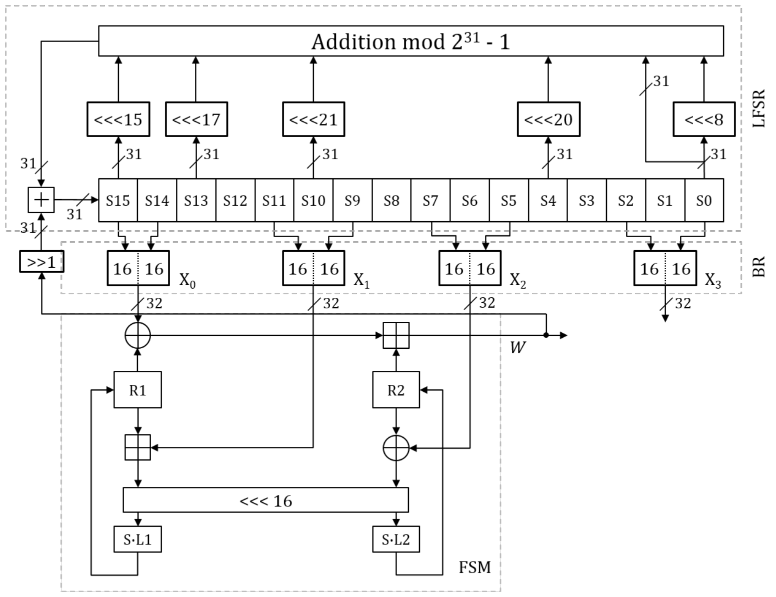
Like SNOW 3G, ZUC has 2 modes of operation:

Initialization mode: during initialization mode the LFSR is initialized using the input key and the IV. The FSM registers are initialized to zero and both the LFSR and the FSM are clocked 32 times. The output of the FSM is shifted right 1 bit to remove the rightmost bit and is then added modulo $2^{31}-1$ with the output of the feedback.

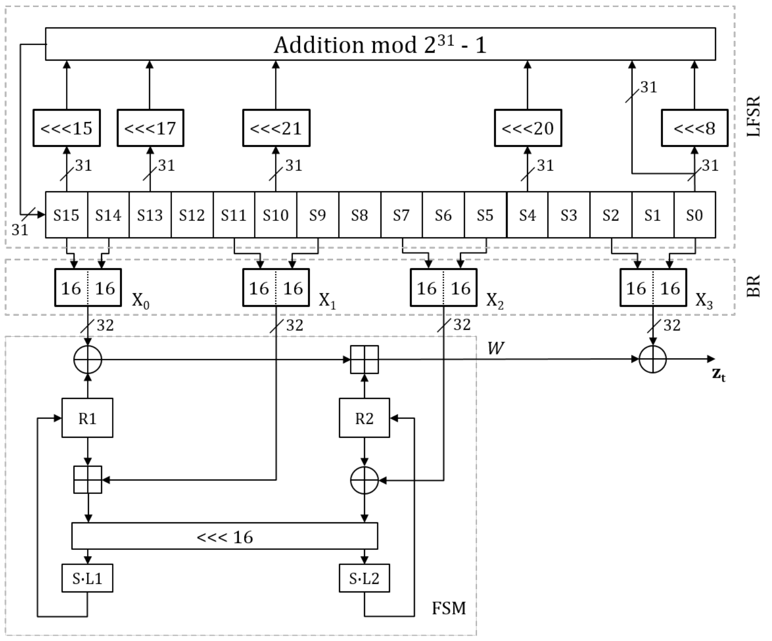
Keystream mode: during keystream mode, the system is clocked once discarding the first output. Then, each time the system is clocked, a 32-bit output from the FSM is produced and XORed with X_3, producing an output word z_t.

LFSR

The LFSR consists of 16 stages of 31 bits, which results in a total of 496 bits. It is initialized during initialization mode with the key and IV as given by [12].


Addition mod 2^{31} - 1
<<<15
<<<17
<<<21
<<<20
<<<8
31
S15 S14 S13 S12 S11 S10 S9 S8 S7 S6 S5 S4 S3 S2 S1 S0
>>1
16 16
X0
X1
X2
X3
32
LFSR
BR
W
R1
R2
<<< 16
S·L1
S·L2
FSM

(a) ZUC during initialization mode


Addition mod 2^{31} - 1
<<<15
<<<17
<<<21
<<<20
<<<8
31
S15 S14 S13 S12 S11 S10 S9 S8 S7 S6 S5 S4 S3 S2 S1 S0
16 16
X0
X1
X2
X3
32
LFSR
BR
W
z_t
R1
R2
<<< 16
S·L1
S·L2
FSM

(b) ZUC during keystream mode

FIGURE 2.10: ZUC ciphering

The *bit reorganization* will take data from the LFSR and produce four 32-bit words X_0, X_1, X_2, X_3 defined as:

1. $X_0 = s_{15H} \| s_{14L}$
2. $X_1 = s_{11L} \| s_{9H}$
3. $X_2 = s_{7L} \| s_{5H}$
4. $X_3 = s_{2L} \| s_{0H}$

Feedback

The feedback performs the function:

$$v = \left[2^{15} s_{15} + 2^{17} s_{13} + 2^{21} s_{10} + 2^{20} s_4 + (1 + 2^8) s_0\right] \; mod \; (2^{31} - 1)$$

Then, this value is input to register s_{15} of the LFSR during keystream mode. During initialization mode it is added modulo $2^{31} - 1$ and then fed to the LFSR. Furthermore, as described in section 10.1.5 of [4], the operation $a \cdot x$ over $GF(p)$, where $a = 2^i + 2^j + 2^k$ can be implemented with cyclic shifts and a modulo p addition as $ax \equiv (x \lll_{31} i) + (x \lll_{31} j) + (x \lll_{31} k) \; mod \; p$.

FSM

ZUC FSM encompasses the following blocks:

- Two 32-bit registers R1 and R2.
- Two S-boxes S0 and S1.
- Two 32-bit XOR.
- Two 32-bit modulo 2^{32} adders.
- Two linear transformations L1 and L2.
- $\lll$ 16 module.

The FSM is defined by the following operations:

1. $W = (X_0 \oplus R_1) + R_2$
2. $W_1 = R_1 \boxplus X_1$
3. $W_2 = R_2 \oplus X_2$
4. $R_1 = S(L_1(W_{1L} \| W_{2H}))$

5. $R_2 = S(L_2(W_{2L} \| W_{1H}))$

The internal registers will be updated as given above. The non-linear substitution box S is a 32×32 S-box composed of 4 juxtaposed 8×8 S-boxes S_0, S_1, S_2, S_3, where $S_0 = S_2$ and $S_1 = S_3$.

The $\lll 16$ module will feed $(W_{1L} \| W_{2H})$ to the $S \cdot L_1$ transformation and $(W_{2L} \| W_{1H})$ to the $S \cdot L_2$ transformation. It performs the concatenation of two 32-bit words, performs a cyclic left shift on the resulting 64-bit word and then splits back into 2 32-bit words.

2.2 Confidentiality Algorithms

In this section, the confidentiality algorithms are presented. The objective of these algorithms is to keep the plain text confidential by transforming the plaintext into a ciphered text that does not bear any meaning to an external entity without the necessary parameters to decode it.

The algorithms are based on the cipher algorithms described in the previous section, and their usage is defined by 3GPP in the technical specifications Security Architecture TS 33.401 [6]. The input parameters of the confidentiality algorithms are a 128 bits confidentiality key KEY, a 32 bits $COUNT$, a 5 bits identity $BEARER$, a 1 bit $DIRECTION$ defining uplink or downlink transmission and a keystream length referred as $LENGTH$. The confidentiality algorithms use the ciphers described in Section 2.1 to produce a keystream of length $LENGTH$ with the aforementioned parameters. The encryption is realized by XORing the plain text with the keystream to result in a confidential text. This allows to perform exactly the same operation for encryption and decryption as long as the keystream is identical.

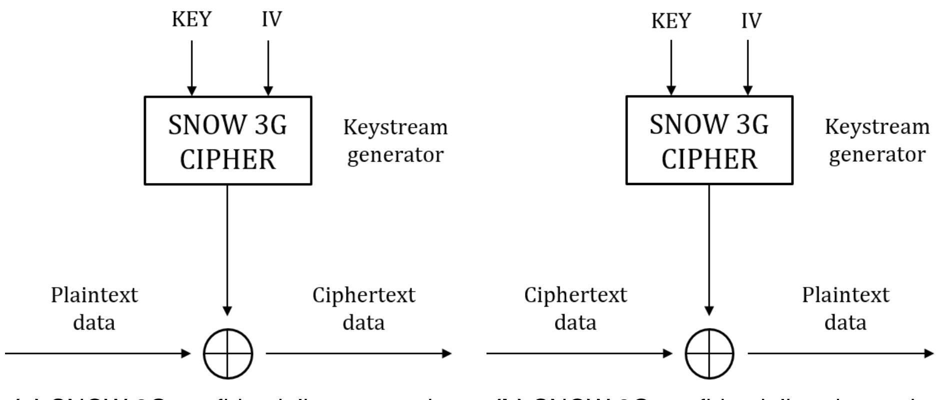
3GPP characterizes 4 EEAs listed here:

EEA0 Null ciphering that provides no security.
128-EEA1 Confidentiality algorithm based on SNOW 3G confidentiality algorithm.
128-EEA2 Confidentiality algorithm based on AES using the Counter (CTR) mode.
128-EEA3 Confidentiality algorithm based on ZUC confidentiality algorithm.

This section introduces each confidentiality algorithm except for EEA0.

2.2.1 128-EEA1: SNOW 3G Based Algorithm

3GPP's 128-EEA1 algorithm is identical to the ETSI/SAGE UEA2 Confidentiality algorithm specified in [7]. Therefore, the keystream produced by the cipher block described in Section 2.1.1 will be used to mask the plaintext data using an XOR operation as shown in Figure 2.11. The confidentiality algorithm will work on 32-bit blocks of data and the IV is constructed as specified in [7].



(a) SNOW 3G confidentiality: encryption **(b)** SNOW 3G confidentiality: decryption

FIGURE 2.11: SNOW 3G confidentiality

2.2.2 128-EEA2: AES Based Algorithm

The AES confidentiality algorithm is based on the AES cipher in CTR mode. This mode defined by NIST in the block cipher modes specification [34]. 3GPP specifies the usage of this recommended mode to generate a keystream with the block cipher AES. Figure 2.12 illustrates the encryption and decryption using the CTR mode.

CTR mode features the production of input blocks passed to the AES cipher (encryption). The returned encrypted blocks are interpreted as a keystream that is XORed with the plain text. The same process is executed on the ciphered text to recover the plain text, using the same generated input blocks. The input blocks $T_1, T_2, \ldots, T_n$, defined as a sequence of 128 bits counter blocks, are built as follows: the most significant 64 bits of the block T_i is constant and constructed as a concatenation of $COUNT$, $BEARER$ and $DIRECTION$, with the Least Significant Bits (LSBs) set to 0. The 64 least significant bits of the block T_i is the value of a 64 bits counter. This counter increments by 1 for every new block, the value being module 2^{64}.

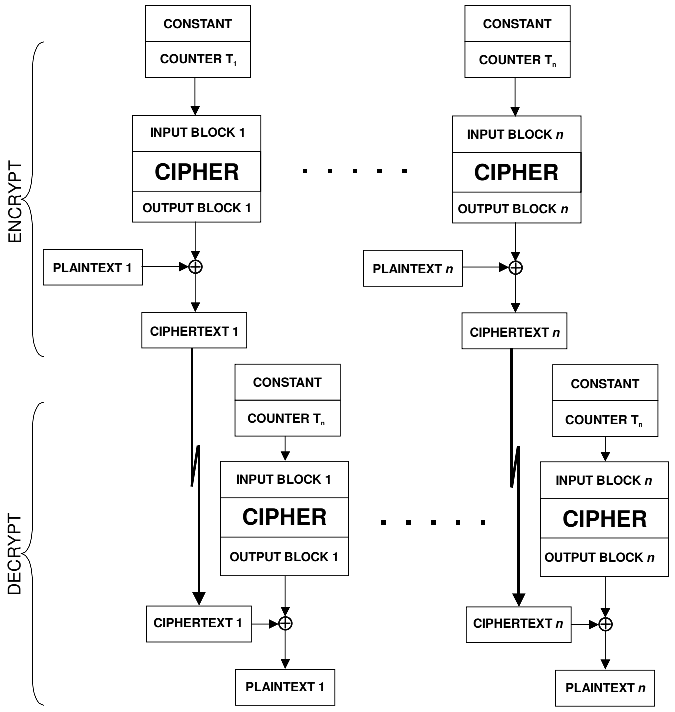


FIGURE 2.12: AES confidentiality using the CTR mode. Source [34]

2.2.3 128-EEA3: ZUC Based Algorithm

This algorithm is specified in [11]. The operation is identical to what is shown in Figure 2.11 but using the ZUC keystream generation as defined in Section 2.1.4 to produce the keystreams.

2.3 Integrity Algorithms

The purpose of these algorithms is to protect the integrity of the information, i.e. to authenticate the text as being the original one without any tampered information.

The algorithm processes the data from which it generates an integrity MAC in the form of a 32 bits word. The integrity of the text is valid when the MAC is compared and identical with the expected one. The input parameters of the integrity algorithms are a 128 bits integrity key KEY, a 32 bits $COUNT$, a 5 bits identity $BEARER$, a 1 bit $DIRECTION$ defining uplink or downlink transmission and a keystream length referred as $LENGTH$. These parameters are identical to the ones defined for confidentiality.

The integrity algorithms use the ciphers described in Section 2.1 to produce a 32-bit MAC with the aforementioned parameters.

3GPP characterizes 4 EIAs listed here:

EIA0 Null integrity that provides no protection. The resulted MAC is set with all zeros and is not checked.
128-EIA1 Integrity algorithm based on SNOW 3G in integrity implementation.
128-EIA2 Integrity algorithm based on AES in Cipher-based MAC (CMAC) mode.
128-EIA3 Integrity algorithm based on ZUC in integrity implementation.

This section introduces each integrity algorithm excepted for EIA0.

2.3.1 128-EIA1: SNOW 3G Based Algorithm

3GPP's 128-EIA1 algorithm is implemented the same way as ETSI/SAGE UIA2 Integrity algorithm, specified in [7].

The IV is constructed as specified in this document, with the only difference being that the 32-bit word $FRESH[0], \ldots, FRESH[31]$ is replaced by: $BEARER[0] \| BEARER[1] \| \ldots \| BEARER[4] \| 0^{27}$. In this notation $FRESH[0]$ and $BEARER[0]$ represent the most significant bits of the words $FRESH$ and $BEARER$.

```
set $D = \lceil \mathbf{LENGTH}/64 \rceil + 1$
set $P = \mathbf{z_1} \| \mathbf{z_2}$
set $Q = \mathbf{z_3} \| \mathbf{z_4}$
set $z_5 = \mathbf{OTP}[0], \mathbf{OTP}[0], \ldots \mathbf{OTP}[31]$
for $i$=0 to $D-3$ set
    $\mathbf{M}_i = \mathbf{MESSAGE}[64i] \| \mathbf{MESSAGE}[64i+1] \| \ldots \| \mathbf{MESSAGE}[64i+63]$
end
set $\mathbf{M}_{D-2} = \mathbf{M}_i = \mathbf{MESSAGE}[64(\mathbf{D}-2)] \| \ldots \| \mathbf{MESSAGE}[\mathbf{LENGTH}-1] \| 0 \ldots 0$
set $\mathbf{M}_{D-1} = \mathbf{LENGTH}[0] \| \mathbf{LENGTH}[1] \| \ldots \| \mathbf{LENGTH}[31]$
```

Algorithm 2.1: Snow integrity parameters

Figure 2.13 describes how the 128-EIA1 algorithm is implemented with its parameters defined in Algorithm 2.1.

The *EVAL_M* function is implemented as described in [7]. The *MUL_GF*(2^{64}) block performs a modulo 64-bit multiplication between Q and $EVAL\_M(M_i,\ P) \oplus M_{D-1}$.

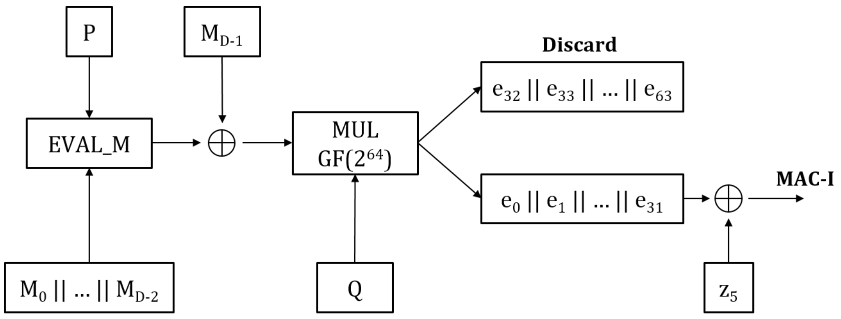
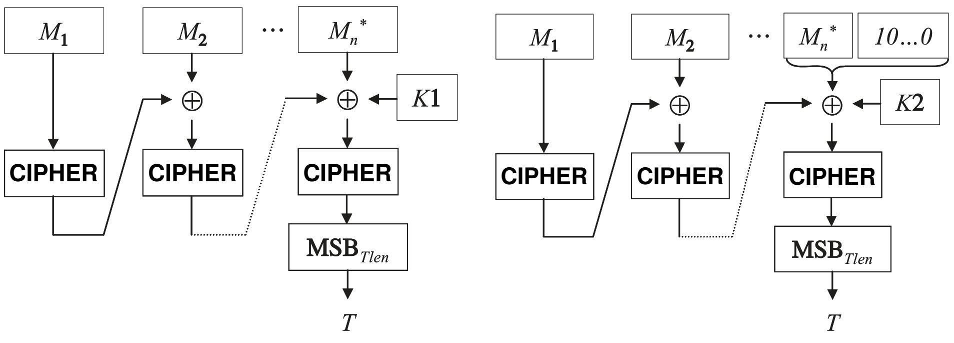


FIGURE 2.13: SNOW 3G integrity diagram

2.3.2 128-EIA2: AES Based Algorithm

The AES integrity algorithm is based on the AES cipher in CMAC mode. This mode is defined by NIST in the block cipher modes specification [35] for authentication. 3GPP specifies the usage of this mode to generate a MAC with the AES cipher. Figure 2.14 illustrates this process using the CMAC mode.

The message data is divided into blocks of 128 bits, which is the input size of data for the AES cipher as described in Section 2.1.2. The resulting data from the cipher is XORed with the next block of the message and encrypted in another call of AES cipher. This is repeated until all the message has been utilized. The 32 Most Significant Bits (MSBs) of the last cipher generation defines the MAC.



(a) without message padding **(b)** with message padding

FIGURE 2.14: AES integrity using the CMAC mode. Source [35]

The input data of the last ciphering is created differently and depends on the message length $LENGTH$. If the final message block M_n^* has a size of 128 bits, i.e. the message length is a multiple of 128 bits, M_n^* is XORed with both the previous cipher output and with the subkey $K1$. Figure 2.14a illustrates the AES integrity algorithm on a message without padding. On the contrary, if $LENGTH$ is not a multiple of 128 bits, then the message is padded with $10\ldots0$ in order to form a complete block, and is then XORed with the previous cipher output and a subkey $K2$. This is showed in Figure 2.14b.

The subkeys $K1$ and $K2$ are constructed from the output L of a ciphering processing input data composed of all zeros. This 128-bit data entirely depends on the integrity key. The following computations determine the value of the subkeys:

```
L← CIPHER(0)
if MSB(L) = 0 then
  | K1← $L \ll 1$
else
  | K1← $(L \ll 1) \oplus R_b$
end
if MSB(K1) = 0 then
  | K2← $K1 \ll 1$
else
  | K2← $(K1 \ll 1) \oplus R_b$
end
```

Algorithm 2.2: Subkey generation in AES integrity

2.3.3 128-EIA3: ZUC Based Algorithm

This algorithm is specified in [11]. The operation of this algorithm to produce an output word **MAC** is shown in Algorithm 2.3.

```
Let $z_i$ be a 32-bit word.
Let $T$ be a 32-bit word.
Let ZUC generate a keystream of
    $L = \lceil \mathbf{LENGTH}/32 \rceil + 2$ words
foreach $i$ in 0 to $32(L-1)$ let
  | $\mathbf{z}_i = \mathbf{z}[i] \| \mathbf{z}[i+1] \| \ldots \| \mathbf{z}[i+31]$;
end
foreach $i$ in 0 to $\mathbf{LENGTH} - 1$
  | if $M[i] = 1$ then $T = T \oplus z_i$;
end
Set $T = T \oplus z_{LENGTH}$

MAC $= T \oplus z_{32\times(L-1)}$
```

Algorithm 2.3: MAC generation in EIA3

Chapter 3

Architecture Exploration

In the previous chapter we observed how the security algorithms are based on certain mathematical operators. AES uses a Rijndael substitution box S_R in the *subbytes* operation, the original SNOW 3G algorithm uses four different lookup tables: Mulα, Divα, S1 and S2 and ZUC uses two non-Rijndael substitution boxes.

In both SNOW 3G and ZUC, different substitution functions are used during the FSM operation and will transform the content of the registers. SNOW 3G has three registers R1, R2 and R3 which will be updated as R1 $\xrightarrow{S1}$ R2 $\xrightarrow{S2}$ R3 whereas ZUC has only two registers R1 and R2 and the next values for these registers will updated as specified in Section 2.1.4.

The following sections will cover the different possibilities when implementing these substitution boxes, the easiest one being a LUT approach. A computation model has been developed for most of the tables and have been compared against a LUT approach.

The first S-box covered is Rijndael box S_R, which is used in both AES and SNOW 3G algorithms. SNOW 3G uses this substitution box internally in the FSM to compute S1. As we will see in the following section, 4 iterations of the S-box will be needed to produce one result of S1. On the other hand, AES must perform the *subbytes* operation on each byte of the state matrix, resulting in a total of 16 calculations. Exploring the possibility of speeding up this computation and/or reducing the area with respect to the matrix implementation is worthwhile and will be covered in the following section.

Later sections will cover the implementation of S-box SQ, the Mulα and Divα operators, and ending with the substitution box S0 used in ZUC algorithm. S-box S1 used in ZUC is very similar in construction to S-box S_R, and thus, the results obtained for S-box S_R are applicable to ZUC S-box S1.

Finally, we will investigate the possibility of integrating SNOW 3G and ZUC. In particular, we will focus on the combination of both FSMs due to their inherent similarities.

The LUTs referred to in this and the following chapters can be synthesized by Catapult HLS to either a multiplexer with constant inputs or a Read-Only Memory (ROM) [27]. Selecting a 65 nm Application-Specific Integrated Circuit (ASIC) synthesis, the tool implements the LUT using a ROM.

The area numbers presented in this and the following sections have been normalized against the smallest design, on a per table basis. Thus, we will compare the relative sizes of the different implementations proposed.

3.1 Rijndael S-box S_R Implementation

The S-box S1, which is used to transform the data available in register R_1 in SNOW 3G ciphering, uses the permutation box as in the Rijndael algorithm. This transformation converts a 32 bit input word w into an output word r, where each four byte word is considered a polynomial over GF_{2^8}, defined by the irreducible polynomial $x^8 + x^4 + x^3 + x + 1$.

The S1 transformation is defined in [20] as follows:

$$\begin{array}{lclclclcl}
r_0 = & (x+1) & S_R(w_3) & + & & S_R(w_2) & + & & S_R(w_1) & + & x & S_R(w_0), \\
r_1 = & & S_R(w_3) & + & & S_R(w_2) & + & x & S_R(w_1) & + & (x+1) & S_R(w_0), \\
r_2 = & & S_R(w_3) & + & x & S_R(w_2) & + & (x+1) & S_R(w_1) & + & & S_R(w_0), \\
r_3 = & x & S_R(w_3) & + & (x+1) & S_R(w_2) & + & & S_R(w_1) & + & & S_R(w_0).
\end{array}$$

Thus, the S-box S_R is used in both AES and SNOW 3G algorithms. Two alternatives have been considered to calculate the result of this substitution box. The first approach is using the substitution matrix provided in [20]. Therefore, each input would be mapped to an output value as defined by the matrix.

The second approach requires more elaboration and consists on an on the fly calculation of the S-box values. As we observe in the S1 transformation above, the Rijndael S-box transformation is applied on each input byte (w_0, w_1, w_2, w_3). This would require 4 different computations of an S_R substitution every time the FSM in SNOW 3G is clocked.

This substitution box is computed in two main steps [33] by:

1. Determining the multiplicative inverse for the input number in Rijndael's field. The element zero ($\{00\}$) has no inverse, and thus will be mapped to itself.

2. Applying an affine transformation over Galois Field, given by:
   $b_i^{'} = b_i \oplus b_{(i+4)mod8} \oplus b_{(i+5)mod8} \oplus b_{(i+6)mod8} \oplus b_{(i+7)mod8} \oplus c_i$.

The affine transformation can be expressed in matrix form as in equation (3.1), where $c = \{1, 1, 0, 0, 0, 1, 1, 0\}$.

$$\begin{bmatrix} b_0^{'} \\ b_1^{'} \\ b_2^{'} \\ b_3^{'} \\ b_4^{'} \\ b_5^{'} \\ b_6^{'} \\ b_7^{'} \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \\ b_4 \\ b_5 \\ b_6 \\ b_7 \end{bmatrix} + \begin{bmatrix} 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \\ 0 \end{bmatrix} \tag{3.1}$$

One of the advantages of using an on the fly implementation is that these operations can be exchanged, performing first an affine transformation and then an inversion in Galois Field computing this way the inverse S_R, used in AES decryption. With the same hardware modules and only a little overhead added by multiplexing logic, two different substitution boxes could be implemented.

The block computing the multiplicative inverse of an element in Galois Field and the block computing the affine transformation as described above have been developed. The inversion algorithm involves multiplication and division over a Galois Field using the irreducible polynomial already mentioned. The implementation of the inverse function is presented in Figure 3.1.

As we observe in the flow chart, the inversion is based on two subfunctions *mulgf* and *divgf*, which correspond to a multiplication and division over a Galois Field respectively. These two functions are detailed in the following sections.

3.1.1 Multiplication in Galois Field 2^8

We wish to multiply two polynomials $f(x)$ and $g(x)$ in a Galois Field of order 2^8, where the irreducible polynomial is $m(x) = x^8 + x^4 + x^3 + x + 1$. The resulting product $f(x) \cdot g(x)$ is reduced modulo $m(x)$ ensuring that the result is always a polynomial of degree less than 8.
Figure 3.2 details how this multiplication is performed. This algorithm is based on the one presented in section 4.2.1 of [33] and defines a subfunction called *xtimes*. This operation on bytes represents a multiplication of the input x by the constant $\{02\}$ and thus, we can use this function to recursively calculate a multiplication by any constant.

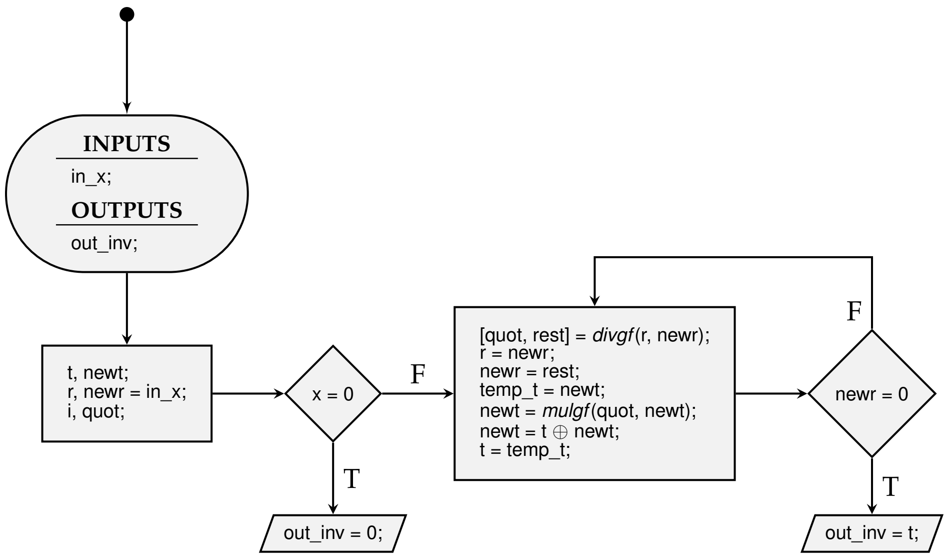


FIGURE 3.1: Flow graph *Inverse(x)* in Galois Field

3.1.2 Division in Galois Field 2^8

The division is implemented as shown in Figure 3.3 and is based on long polynomial division.

The function *ld_ones* will return the position of the first '1' of the data input. The function *getRest(vect_in, start, end)* will return a slice of the input vector between start and end, and is used to get the rest in a polynomial division.

3.1.3 Implementation Results and Comparison

Once the on the fly models for Rijndael S-box have been developed and tested, the LUT approach has been compared to the computation approach. Since the on the fly computation is able to calculate both direct and inverse S_R substitution values, it must be compared with the results obtained implementing two 8×8 look-up tables.

As we described in the previous sections, the maximum number of iterations in the main loop governing the inverse, multiplication and division functions is 8 iterations in each one of them. Catapult HLS will make it possible to explore the different options we have when implementing the S-box calculation. Other solutions which take into account partially unrolled hardware resources can be obtained, however as we observe in Table 3.1, the results provided by a LUT approach are superior.

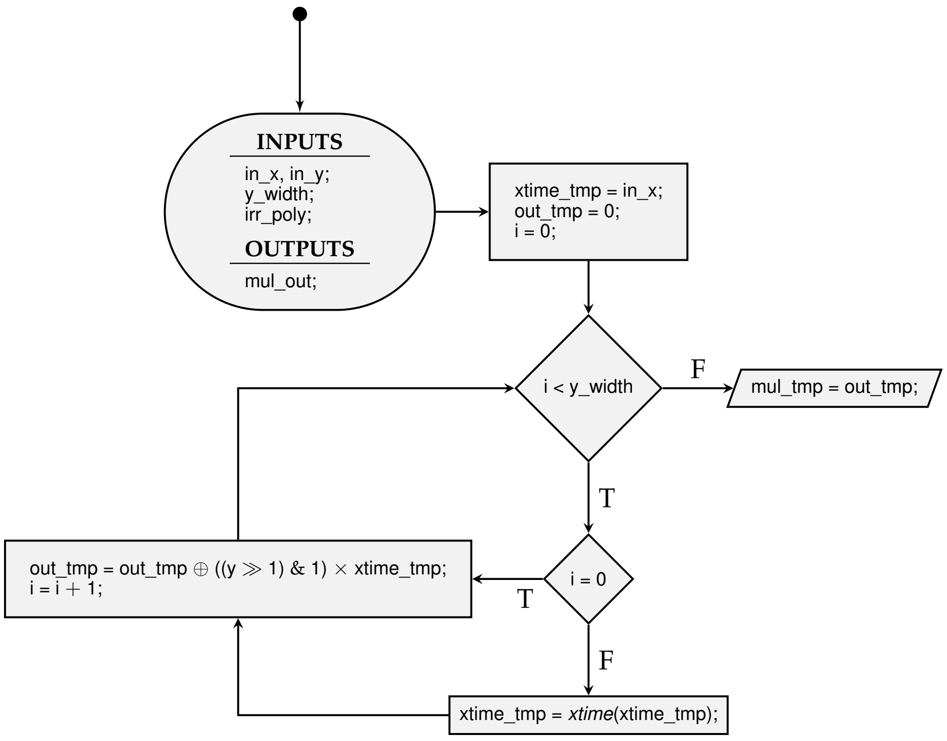


FIGURE 3.2: Flow graph *Mul(x, y)* in Galois Field

TABLE 3.1: Comparison of different implementations of S-box S_R

Implementation	**Area (normalized)**	**Throughput (cycles)**	**Slack @ 50 MHz (ns)**
Solution 1	7.3	8	4.88
Solution 2	5.9	64	15.23
Solution 3	6.1	128	15.21
Solution 4	250.0	120	15.66
Solution 5	1.0	1	19.76

Solution 1	*invgf* kept rolled, *mulgf* and *divgf* unrolled.
Solution 2	*invgf* and kept *divgf* rolled, *mulgf* unrolled.
Solution 3	all loops rolled.
Solution 4	*invgf* unolled, *mulgf* and *divgf* rolled.
Solution 5	LUT approach.

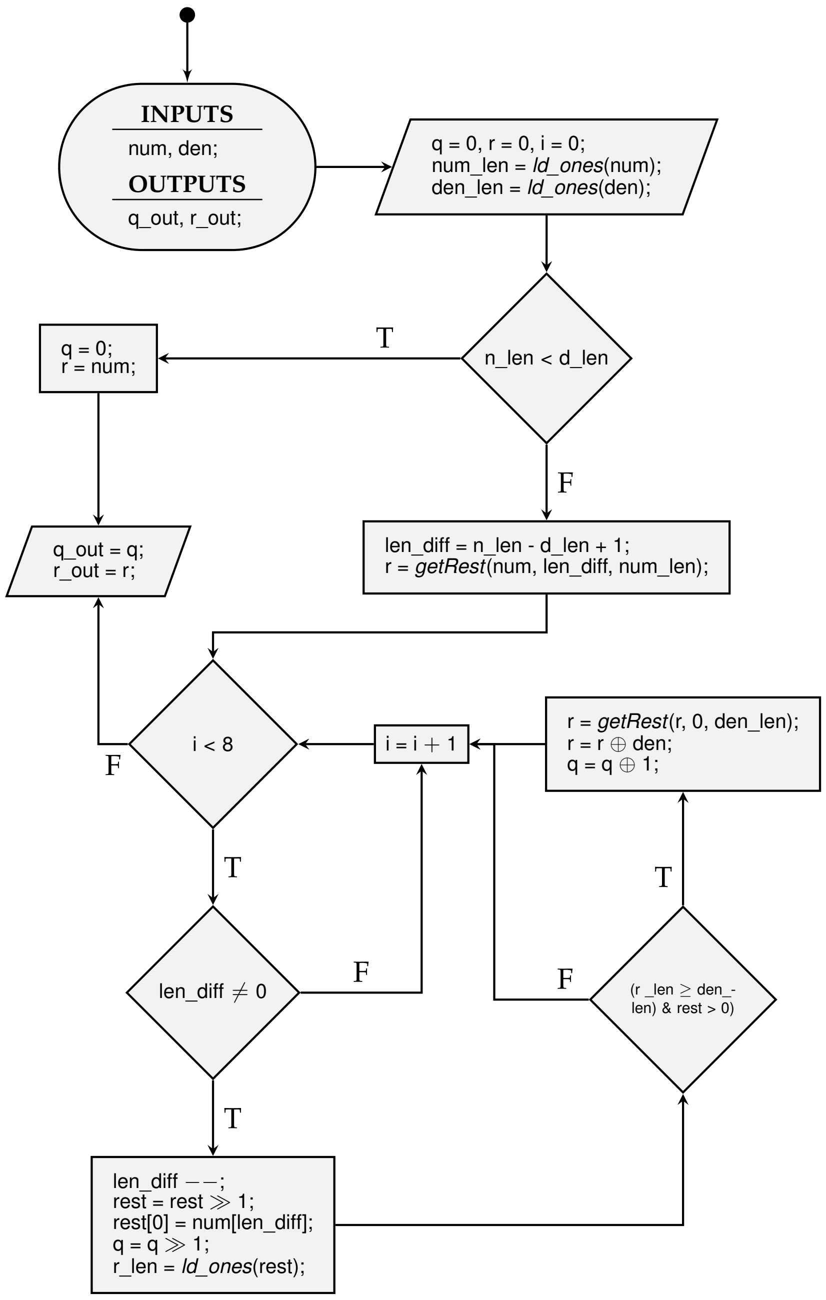


FIGURE 3.3: Flow graph *Div(x, y)* in Galois Field

The conclusion drawn from this architectural exploration is that the complexity of the functions involved in the S-box S_R calculations makes this particular S-box unsuitable for an on-the-fly calculation.

3.2 S-box S_Q Implementation

This substitution box is used only in SNOW 3G algorithm to perform the S2 transformation, and is used in a similar way than S-box S_R to calculate S1. Considering a 32 bit input word w where the bytes are in this transformation interpreted as elements of a Galois Field 2^8 defined by the irreducible polynomial $x^8 + x^4 + x^3 + x + 1$, the output of S2 will be given by the following relations:

$$\begin{array}{lllllll}
r_0 = & (x+1)\ S_Q(w_3) & + & S_Q(w_2) & + & S_Q(w_1) & + & x\ S_Q(w_0), \\
r_1 = & S_Q(w_3) & + & S_Q(w_2) & + & x\ S_Q(w_1) & + & (x+1)\ S_Q(w_0), \\
r_2 = & S_Q(w_3) & + & x\ S_Q(w_2) & + & (x+1)\ S_Q(w_1) & + & S_Q(w_0), \\
r_3 = & x\ S_Q(w_3) & + & (x+1)\ S_Q(w_2) & + & S_Q(w_1) & + & S_Q(w_0).
\end{array}$$

The S-box SQ is constructed using the Dickson polynomial:

$$g_{49}(x) = x \oplus x^9 \oplus x^{13} \oplus x^{15} \oplus x^{33} \oplus x^{41} \oplus x^{45} \oplus x^{47} \oplus x^{49}$$

Then, the output will then be given by: $S_Q(x) = g_{49}(x) \oplus 0x25$.

The architectural diagram for this algorithm is shown in Figure 3.4. Internally, *mulgf* is being used to perform a multiplication in GF. Note that the irreducible polynomial used in this calculation is different from the one used for S-box S_R.

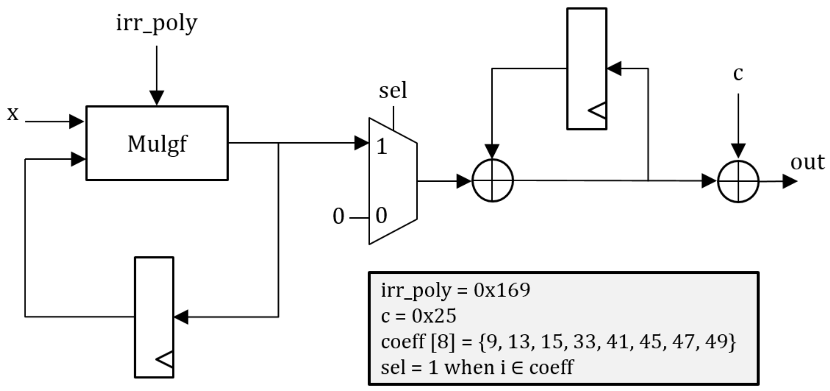


FIGURE 3.4: S-box S_Q simplified hardware diagram

The results obtained from the different implementations are shown in Table 3.2.

The different solutions are constructed in a similar manner than we did for the S-box S_R and are detailed below. The on-the-fly solution with best results is solution 2, in which all the 8 iterations of the *mulgf* loop are unrolled and the Dickson

TABLE 3.2: Comparison of different implementations of S-box S_Q

Implementation	Area (normalized)	Throughput (cycles)	Slack @ 50 MHz (ns)
Solution 1	19.0	1	10.96
Solution 2	2.2	49	18.32
Solution 3	43.2	384	18.91
Solution 4	2.5	392	18.18
Solution 5	1.0	1	19.76

Solution 1	Both loops unrolled.
Solution 2	Dickson polynomial loop kept rolled, *mulgf* unrolled.
Solution 3	Dickson polynomial loop unrolled, *mulgf* kept rolled.
Solution 4	Both loops rolled.
Solution 5	LUT approach.

polynomial loop, which is a simple loop that will perform conditional XOR at certain iterations, is kept rolled. The Dickson polynomial can be partially unrolled while maintaining the *mulgf* loop completely unrolled to achieve better throughput results at the expense of a bigger area.

Catapult shows that better results are provided for a LUT approach with respect to its on-the-fly counterpart.

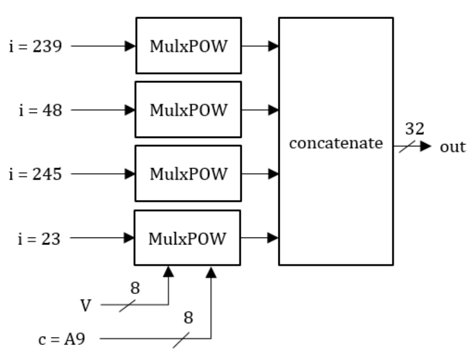
3.3 Mulα and Divα Implementation

The Mulα and Divlα operators are used in SNOW 3G feedback loop as was seen in Section 2.1.1. As described in this section, these operators map an 8 bit input to 32 bit output. The output is computed using four instances of the function *MulxPow* operation as shown in Figure 3.5.

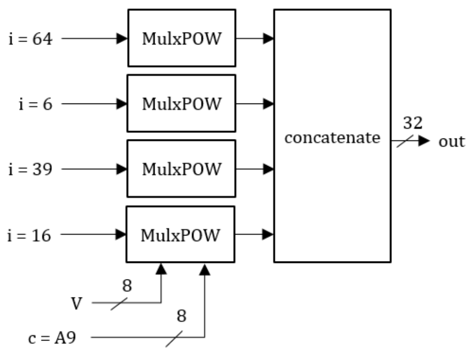
These four different instances will have the same inputs $(V,\ c)$ but will differ on the value of input i. The input i will control the number of times the inner function *mulx* is executed. Therefore, *MulxPow* is constructed by recursively calling the operator *mulx*, and the number of times it is called is controlled by the control variable i.

Figure 3.7 illustrates the different implementations of Mulα and Divα explored. More information about the different solutions can be found in Table 3.3.

Solution 4 is depicted in Figure 3.8. We observe how a fully unrolled architecture

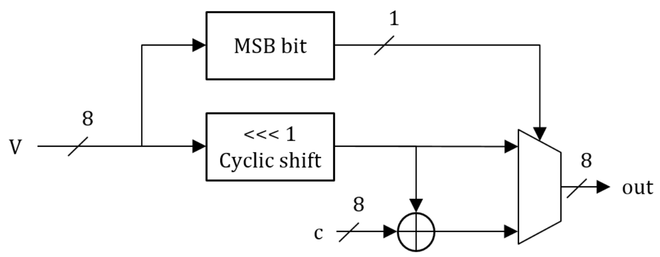


(a) Mulα

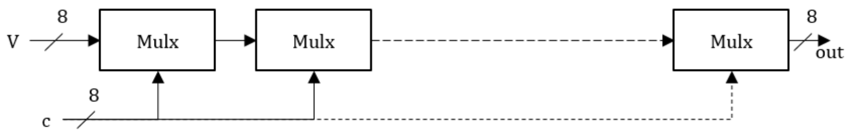


(b) Divα

FIGURE 3.5: Mulα and Divα implementation



(a) Mulx



(b) MulxPow

FIGURE 3.6: Mulx and MulxPow

with 4 pipeline stages presents an area overhead of 51.2 % with respect to the most area efficient achieved solution, number 5 in the table. The pipelined architecture will have a latency of 4 clock cycles, but after that one output will be produced every clock cycle. Solution 5 combines multiple optimizations in order to achieve the lowest area.

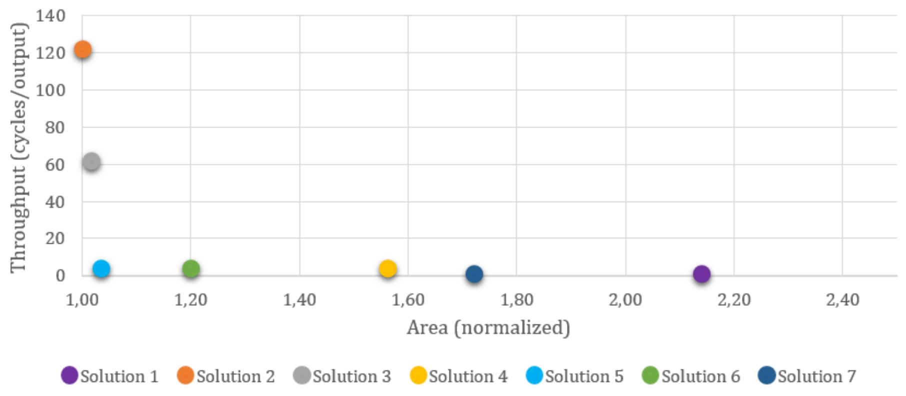


FIGURE 3.7: Different implementations of Mulα and Divα

TABLE 3.3: Comparison of different implementations of combined Mulα and Divα

Implementation	**Area (normalized)**	**Throughput (cycles)**	**Slack @ 50 MHz (ns)**
Solution 1	2.1	1	2.37
Solution 2	1.0	123	19.03
Solution 3	1.0	62	18.91
Solution 4	1.6	4	10.6
Solution 5	1.0	4	15.43
Solution 6	1.2	4	10.97
Solution 7	1.7	1	19.76

Solution 1 Fully unrolled architecture, no pipelining.
Solution 2 One pipeline stage.
Solution 3 Two pipeline stages.
Solution 4 Four pipeline stages.
Solution 5 Multiple optimizations combined.
Solution 6 Fully merged solution with four pipeline stages. Assumes Mulα and Divα have the same input.
Solution 7 LUT approach.

3.4 ZUC S-box S0 Implementation

The implementation of S-box S0 in ZUC algorithm follows the specification in [4]. The 3 transformations P1, P2, P3 in Figure 3.9, are transforms over GF(16) and are

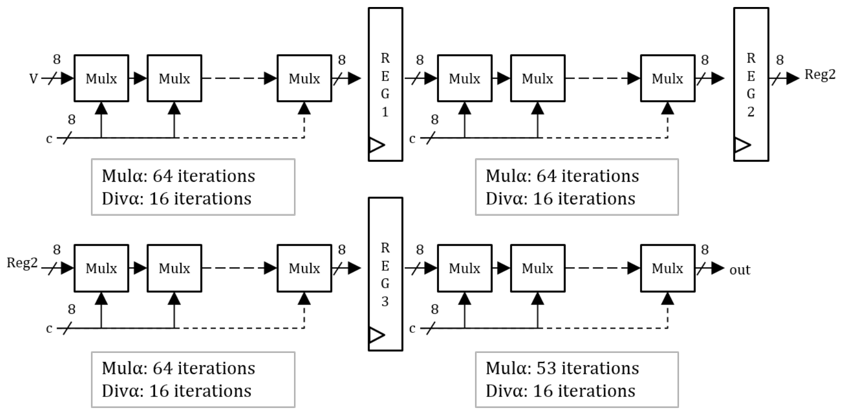


FIGURE 3.8: Alpha operators pipelining

detailed in the specification.

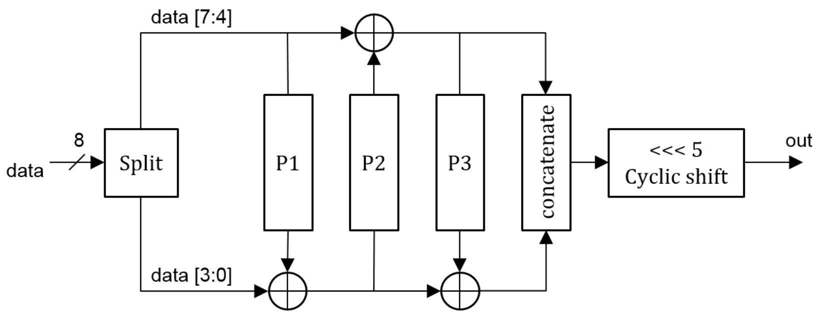


FIGURE 3.9: ZUC S-box S0 implementation

TABLE 3.4: Comparison of different implementations for ZUC S-box S0

Implementation	Area (normalized)	Throughput (cycles)	Slack @ 50 MHz (ns)
Solution 1	1	1	18.96
Solution 2	1.7	1	19.76

Solution 1 On-the-fly implementation.
Solution 2 LUT implementation.

The on-the-fly implementation of ZUC S-box S0 uses less hardware resources than the LUT approach.

3.5 SNOW 3G and ZUC Combination

SNOW 3G and ZUC are both stream cipher algorithms which show clear similarities in structure. In this section we will focus on the results obtained when trying to combine some of the main blocks in these algorithms.

3.5.1 Combined LFSR

The LFSR is a module that can be easily merged between both algorithms, although it presents a slight difference: ZUC uses 31-bit registers while SNOW 3G uses 32-bit registers. Also, ZUC includes a reorganization stage that takes data from the LFSR and feeds it to the FSM.

3.5.2 Combined Feedback

The structural differences in the feedback logic between both algorithms makes this module unsuitable for combining. Having two separate modules with a multiplexed output would be the desired solution in this case.

3.5.3 Combined FSM

From Section 2.1, we observe that SNOW 3G and ZUC could share two 32-bit registers, two 32-bit modulo adders and two 32-bit XORs. A Combined FSM has been implemented and compared against individual FSMs. The multiplexing overhead in this implementation is five 32-bit multiplexers which, as we observe in Table 3.5, leads to a greater area footprint than having individual FSMs.

TABLE 3.5: Comparison between individual and combined FSM implementation

Implementation	Area (normalized)	Throughput (cycles)	Slack @ 50 MHz (ns)
SNOW 3G	1.2	4	17.85
ZUC	1	4	16.22
Total	**2.2**	**4**	**16.22**
Combined	**2.4**	**4**	**15.35**

As a remark, the results provided in Table 3.5 include the substitution boxes, S1-S2 for SNOW 3G, and S0-S1 for ZUC. All the S-boxes are implemented as LUTs expect for ZUC S-box S0. Only one S-box of each kind is implemented leading to a FSM that produces one output word every 4 clock cycles.

Chapter 4

Crypto Processor

In this chapter, the complete hardware security Intellectual Property (IP) architecture is presented. The module is composed of processing elements that realize the functionality defined by the algorithms detailed in Chapter 2, and of a interconnect logic used to create a data and control path between the processing elements. The module also contains the control and interface to be used and connected to an AMBA High-performance Bus (AHB). This allows the module to receive and send data while being configured by a master, a Central Processing Unit (CPU) or a micro-controller for example.

A top-down hierarchy is used to present the hardware implementation, starting from the top overview with its Interface (IF) and data path. Furthermore the controller and the processing cores are introduced.

4.1 Top Overview

The security IP is designed as a standalone block to implement all the hardware to secure the data as defined by 3GPP for NB-IoT. The module includes the ciphering logic to process the three algorithms used for NB-IoT, SNOW 3G, AES and ZUC; the blocks realizing confidentiality and integrity cipher mode; the multiplexing logic between the different algorithms and modes; and the interfacing logic as a Slave on an AHB to receive and send as well data as parameters.

The top level architecture is illustrated in Figure 4.1. The processing elements composing the security IP are represented in green with the data path connection and blocks in black. The parameters received and stored in the AHB IF and used to configure the blocks are represented in blue.

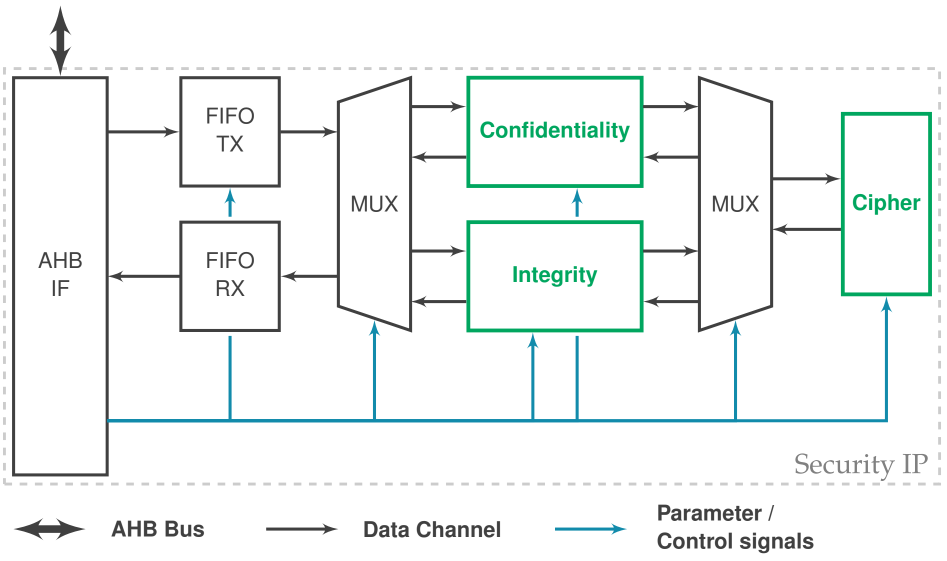


FIGURE 4.1: Block diagram of Security IP top

4.1.1 AHB Interface

The security IP communicates only via an AHB IF. The AHB is a bus protocol part of Advanced Microcontroller Bus Architecture (AMBA) created by ARM as an open-standard for on-chip communication between hardware blocks. The bus complete specification is available on ARM website [14]. The version used is AHB-Lite from ARM AMBA 3 standard.

AHB separates the address and its associated data on two distinct cycles, called address and data phase respectively. A cycle is composed of the address phase and the data phase associated to the previous address phase. This overlapping of address and data phases enables a high performance operation by ensuring a pipelined access to the bus. This is illustrated in Figure 4.2. For example at the time t_2, the address cycle is A_2 while the data cycle is D_1.

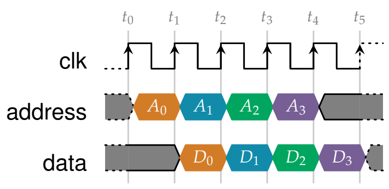


FIGURE 4.2: Timing diagram of AHB address/data cycles

The Slave samples the transmission request sent by the Master on the address phase and defined by the following signals:

hsel Select the Slave when active high. The Slave must ignore the transaction when the signal is low.

htrans Select the type of transaction. This 2-bit signal possesses four values specifying the type:

IDLE This type indicates that the Master performs no data transfer. The Slave must ignore the transfer.

BUSY This type allows the Master to insert idle state during a burst. The burst sequence is not stopped but the Master is unable to provide the next transfer immediately.

NSEQ This type indicates a single transfer of the first transfer of a burst sequence.

SEQ This type indicates that the following transfers are part of a burst. In a burst, the address is related to the address of previous transfer.

haddr Value of the address of the transaction. The Slave uses the address to select which data is addressed by the request.

hwrite Select a write transaction when the signal is high, else a read transaction is enabled.

hburst This signal contains the 3-bit value indicating the type of burst of the transfer.

hready Set by the Slave to indicate the Master that the transaction is acknowledged when the signal is high.

hresp This signal provides the Slave's response to the Master, indicating an error when high.

The transaction data is present on a following phase and composed of these signals:

hrdata Data received by the Master from the Slave.

hwdata Data sent by the Master to the Slave.

The Slave samples all signals on the address phase while only the two data signals, *hrdata* and *hwdata*, are sampled during the data phase. The address and the data signals have a size of 32 bits in this design. The Slave will only be capable of requesting a data phase extension by setting *hready* to zero.

As shown in Figure 4.3 that illustrates an AHB write transaction, during the cycle t_1 the Master asserts the address to A_0. The write transaction is defined with the active high *hwrite* signal during the address cycle. The recipient Slave is selected with its *hsel* signal enabled and it provides its acknowledge to the Master with the signal *hready*, high showing that the Slave is ready during the cycle t_1. The Master provides the data D_0 associated to address A_0 on the next cycle t_2. On the same cycle, it also provides the address A_1 for the next transfer. The data D_1 set during the cycle t_3 is associated to a signal *hready* placed low by the Slave. The Master must retain constant the data value D_1 on the *hwdata* signal as well the other signals, like the address A_2 of the next transfer, until the Slave is capable of accepting the current request by setting the *hready* signal to logic '1'. This is done in cycle t_6.

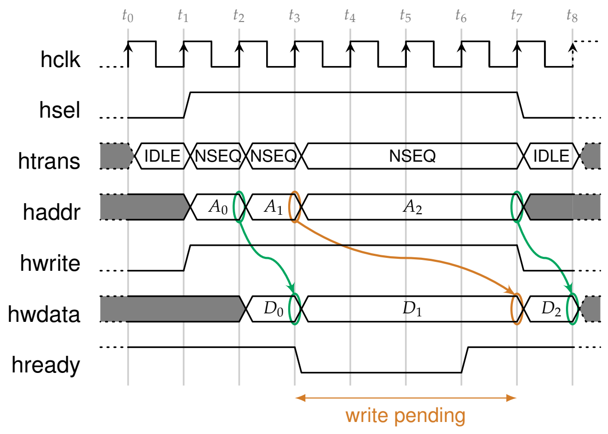


FIGURE 4.3: Timing diagram of AHB write

The read sequence is relatively equivalent to the write transaction previously described. The signal *hwrite* is set to 0 to define a read transaction. The data is present on the *hrdata* signal instead of *hwdata*. The data is asserted by the Slave on the same cycle as it activates its ready status with an *hready* active high. Depending on the Slave, the data may be directly available or after some time. In Figure 4.4 the two cases are illustrated. At cycle t_2, the address A_0 is received by the Slave IF, the signals *hsel* and *hready* being both active high. On the next cycle, t_3, the data has been transmitted to the Master as the signals *hsel* and *hready* are still active high. This is the fastest answer possible on AHB. At cycle t_4, the Slave deactivates its ready signal, meaning that the previous read requested data is not ready and therefore extending the current data phase. The Master must keep constant the address cycle till the Slave asserts the *hready* signal. This is done at time t_7 when the Slave is able to provide the read data.

The Security IP treats the burst transfer as normal transfer. As the module uses an unique address for the data, specified in Table 4.1, transferring multiple data is not realized with burst. This type of transfer defines an incrementing address related to the previous cycle. In the case of this IP, the transfer is always *NSEQ* for the data. Nevertheless, for the register-mapped parameters, a sequential access may be done with a burst.

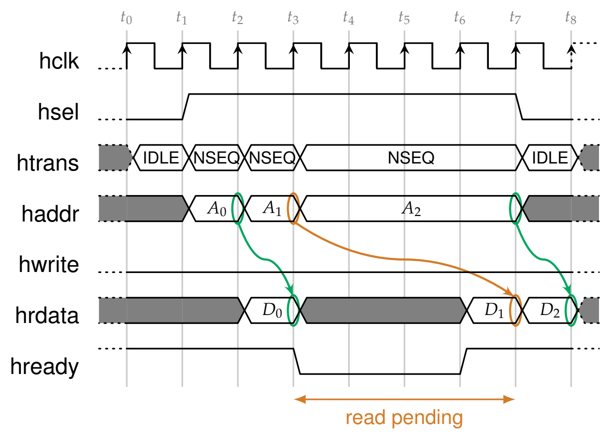


FIGURE 4.4: Timing diagram of AHB read

4.1.2 Data Path

The data path regroups the signals where the data transits between the connected blocks. In order to keep the data synchronized between modules accepting different rate and timing for data, handshake signals are adopted. This method allows the data to be passed only when the send and receiver are synchronous, both of them ready at the same cycle. A channel names the collection of data with handshake signals.

Figure 4.5 presents handshake signals accompanying the data. The signals are defined from the sender point of view. *data_out* and *valid_out* are asserted by the sender while *ready_in* is set by the receiver and collected by the sender. A data is sent when it is acknowledged, i.e. both *valid_out* and *ready_in* signals are active high. For the cycles t_2, t_3 and t_4, the handshake is immediately done as no delay exists between the cycle. During the cycles t_5 and t_6, the transfer is pending for the receiver as it set its ready signal to logic '0'. The sender must keep its outputs constant, in this example with the data D_3 and the signal *valid_out* high. The acknowledgment is realized at the cycle t_7 when *ready_in* is high. The next transfer shows the case with the receiver ready and waiting for a transaction from the sender. The data provider takes three cycles to present a new data, the signal *valid_out* being low during the cycle t_8 and t_9.

4.1.3 Channel Multiplexing

Previously in Section 2 was defined the confidentiality and integrity mode. As these elements are mutually exclusive, their data paths are multiplexed using channel multiplexers. Two types of multiplexing exist: 2–to–1 channel multiplexer and 1–to–2, the latter also being called demultiplexer. Each block MUX

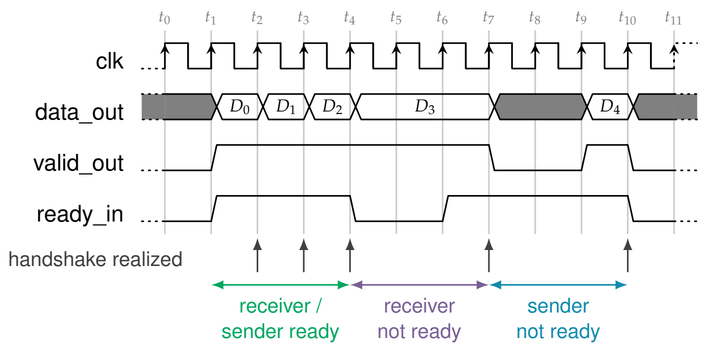

t_0 t_1 t_2 t_3 t_4 t_5 t_6 t_7 t_8 t_9 t_{10} t_{11}
clk
data_out
valid_out
ready_in
D_0 D_1 D_2 D_3 D_4
handshake realized
receiver / sender ready
receiver not ready
sender not ready

FIGURE 4.5: Timing diagram of data handshake

illustrated on the block diagram of the Security IP top in Figure 4.1 is composed of one 2–to–1 channel multiplexer and one 1–to–2 channel demultiplexer. During each security process, only one data path is defined by these MUX blocks: via the confidentiality processing element or via the integrity one. This is selected with a 1 bit signal *i_integrity_en* driven by the control module. The first MUX block connects the TX and RX First-In, First-Out (FIFO) memories to the appropriate processing element, confidentiality or integrity core. The second MUX block multiplexes the channels signals from the confidentiality and integrity cores to the ciphering module.

4.1.4 FIFOs

FIFOs act as a buffer on the data path. In order to not limit or be limited by the AHB and its IF, the FIFO containing a memory allows to store a larger amount of data in the IP. The FIFO size was implemented to 16 words of 32 bits. The input FIFO TX takes one parameter from the controller, the data length in number of 32-bit words. This is explained by a specific signal part of the data channel path connecting from the AHB IF to the cipher core. Each of the processing elements uses a signal *data_last* to correctly conclude their processing (this is detailed in Chapter 2. This signal is generated by FIFO TX which counts the number of data it provides and compares this value to the data length provided by the controller. The FIFO uses data channels as input and output. The data handshake protocol allows to avoid overflow and underflow of the FIFO. It guarantees no loss of data both in writing and reading.

4.2 Controller

The controller is present in the AHB IF module illustrated in Figure 4.1. This component encloses the memory mapped registers storing all the security IP parameters and control capabilities. It also interfaces the AHB IF to the data path via channel IF.

4.2.1 Memory Mapped Registers

The security IP incorporates several registers mapped on the AHB. They handle the parameters and status of the IP. The CPU or microcontroller software can access this information to configure the ciphering process. Table 4.1 lists the existing registers, their respective address, write and/or read access (Read-Write (RW), Read-Only (RO) and Write-Only (WO)), as well as a description of their content.

TABLE 4.1: List of security IP register

Register	Address	Access	Description
VERSION	0x00	RO	Contains the IP hardware version.
CONTROL	0x04	WO	Write the configuration bits of the IP.
STATUS	0x04	RO	Read the status bits of the IP.
DATA_LENGTH_LSB	0x08	RW	Contains the LSB word of the data length.
DATA_LENGTH_MSB	0x0C	RW	Contains the MSB word of the data length.
KEY_WORD_0	0x10	RW	Contains the word 0 of the key.
KEY_WORD_1	0x14	RW	Contains the word 1 of the key.
KEY_WORD_2	0x18	RW	Contains the word 2 of the key.
KEY_WORD_3	0x1C	RW	Contains the word 3 of the key.
CONTEXT_WORD_0	0x20	RW	Contains the word 0 of the context.
CONTEXT_WORD_1	0x24	RW	Contains the word 1 of the context.
CONTEXT_WORD_2	0x28	RW	Contains the word 2 of the context.
CONTEXT_WORD_3	0x2C	RW	Contains the word 3 of the context.
TX_DATA	0x30	WO	Write the data to be processed by the cipher.
RX_DATA	0x30	RO	Read the data to processed by the cipher.

The Table A.1 present in Appendix A details the bit organization of the *CONTROL /STATUS* register.

4.2.2 FIFO Channels Interface

The address of the register *TX_DATA* and *RX_DATA* maps the AHB to the FIFOs channel interface. Contrary to a register mapped, the data is not always available for writing or reading with FIFO. When a transfer is requested at this address, a request signal is set, either *fifo_wr_transfer* or *fifo_rd_request* depending on the *hwrite* signal. The write request is transmitted as the valid signal of the data channel to the *FIFO TX*. If the FIFO is not full, its handshake signal ready is high. This enables the *hready* on the AHB side while the write request signal *fifo_wr_-transfer* is de-asserted. When the FIFO is full, it asserts the ready signal of the channel to '0'. In this case, the write request remains valid and the ready signal on the bus *hready* is set to '0'. The AHB is blocked with this pending request. The safeguard would be to enable a counter. After reaching a defined timeout value, it would cancel the transfer request and send an error respond to the Master via the signal *hresp*. This is not implemented in the current version of the Security IP. The interface for reading the *FIFO RX* uses the same principle. The differences are the connections with the channel signals, where *fifo_rd_transfer* is linked to the FIFO ready and the data valid coming from the FIFO is used for the response on the AHB.

4.3 Cipher Block

The cipher block contains the SNOW 3G, AES, and ZUC cipher modules. The modules are used as keystream generation cores that will feed a keystream to the confidentiality or integrity block, depending on the *i_integrity_en* signal. Depending on the value of the *i_algo_sel* signal, the cipher block with route the input *i_start* to the appropriate cipher module. Then, the cipher block will provide a keystream from the selected algorithm depending on the value of the *i_algo_-sel* signal. This section describes the hardware architecture of those algorithms' implementation.

4.3.1 SNOW 3G Cipher Block

This section describes the implementation of SNOW 3G algorithm as described in Section 2.1.1. Figure 4.6 sketches the FSM that controls the operation of this hardware block.

As was mentioned in Section 2.1.1, SNOW 3G has two different modes of operation: initialization mode and work mode. These are shown on the right hand side of Figure 4.6. The block is initialized in *IDLE* state, signaling a *o_cipher_rdy* = '1', meaning that the block is ready to cipher data. When a *i_start* signal is received, the block goes to *INIT_MODE* state. During this transition, the block initializes

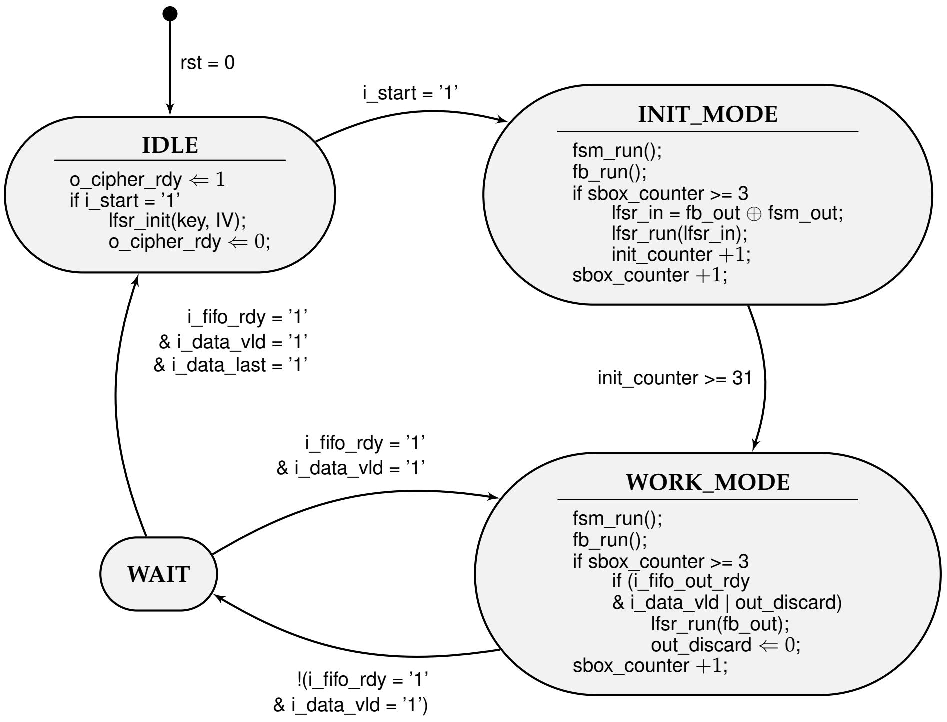


FIGURE 4.6: FSM of SNOW cipher block

the LFSR with the values given by the inputs *i_key* and *i_IV* and puts the *o_cipher_rdy* signal to zero indicating this way a busy state to the crypto processor's control unit. The two 128-bit signals, *key* and *IV*, are stored in internal registers of the aforementioned control unit and are connected to the cipher block. Furthermore, the registers in the FSM are initialized to zero.

Once the cipher block is in initialization phase, it must be clocked 32 times before going into *WORK_MODE*. During this state, the block will first discard the first output, as indicated by the *out_discard* flag. Then, SNOW 3G cipher will produce valid keystream outputs.

In Section 3 we analyzed different possible implementations of the S-boxes S_R and S_Q, which are used internally in the FSM the update the registers R1, R2 and R3. The conclusion in this chapter was that one S-box S_R and one S-box S_Q should be implemented, both as a ROM. The reason for this was that the on-the-fly implementations did not offer any increase in performance of any kind, and also accounted for a bigger area footprint. With this scheme, the FSM can produce one new output every 4 clock cycles. Furthermore, the use of 4 ROMs would account for an unnecessary increase in area.
We recall from Section 3.1 that the S-box S1 will map a 32-bit input w to a 32-bit

output r. This way the content of register R1 will be transformed and stored in register R2: R1 $\xrightarrow{S1}$ R2. How this is done is detailed in Section 3.1 and repeated below for convenience:

$$\begin{array}{llllllll}
r_0 = & (x+1) & S_R(w_3) + & & S_R(w_2) + & & S_R(w_1) + & x & S_R(w_0), \\
r_1 = & & S_R(w_3) + & & S_R(w_2) + & x & S_R(w_1) + & (x+1) & S_R(w_0), \\
r_2 = & & S_R(w_3) + & x & S_R(w_2) + & (x+1) & S_R(w_1) + & & S_R(w_0), \\
r_3 = & x & S_R(w_3) + & (x+1) & S_R(w_2) + & & S_R(w_1) + & & S_R(w_0).
\end{array}$$

In total, six additional registers have been created to store intermediate look-up results. We observe that in order to calculate the output r it will be necessary to store $S_R(w_0)$, $S_R(w_1)$ and $S_R(w_2)$. Then, on the last clock cycle $S_R(w_3)$ is computed and together with the stored values the output word r is generated. The same is done for S-box S_Q which is used to transform the content of R2 and store it in R3: R2 $\xrightarrow{S2}$ R3.

The feedback path has been implemented in a time multiplexed manner which demonstrated, as shown in Section 3, to present optimal results. The FSM has a throughput of 1 output every 4 clock cycles, and therefore is seems natural to use a feedback implementation that also provides an output every 4 clock cycles.

To keep the design synchronized a counter named *sbox_counter* is used. This modulo 4 counter will indicate when a new output word from the FSM is available. Once this output is produced, it may be the case that a component connected to the SNOW 3G keystream generator is not ready to send and/or receive data. In this case, the block will go into *WAIT* state until the sender and receiver blocks are ready to send and receive data, respectively. This situation is met when the handshake signals *i_data_valid* and *i_fifo_out_rdy* are both logic '1'.

Since the LFSR, the feedback, and the FSM all have to be clocked at the same time, the throughput of the design will be given by the following expression:

$$T_{SNOW\ 3G} = max(LAT_{FSM}, LAT_{LFSR}, LAT_{FEEDBACK}) = 4\ clock\ cycles$$

In this scenario, SNOW 3G cipher has an initial latency of:

$$L_{SNOW\ 3G} = 33 \times 4 + 4 = 136\ clock\ cycles$$

to produce the first valid output keystream. Of these, 132 clock cycles are taken by the initialization phase, 4 clock cycles are needed to discard the first output and finally 4 additional cycles to produce the first output.

4.3.2 AES Cipher Block

This block implements the algorithm AES describes in Section 2.1.2. The module is built around a FSM that follows the Algorithmic State Machine with Datapath

(ASMD) illustrated in Figure 2.3. Each of the FSM states, showed in Figure 4.7, corresponds to a transform or function defined by the algorithm.

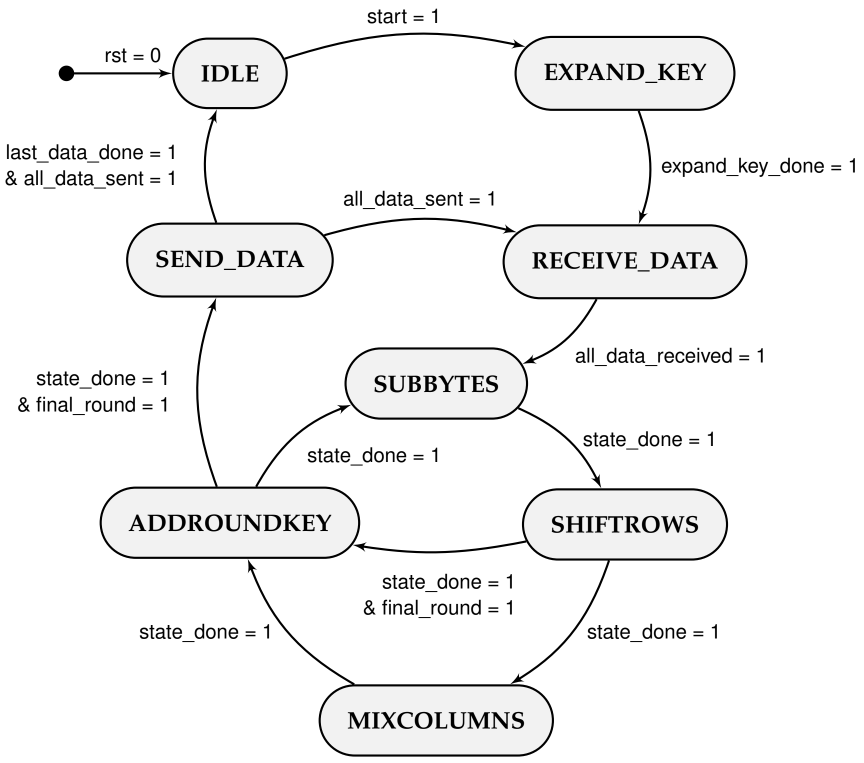


FIGURE 4.7: FSM of AES cipher block

EXPAND_KEY

This hardware generates the expanded key. As the total size of this key is 1408 bits, it is implemented as a Read-Access Memory (RAM) with 44 words of 32 bits. The usage of a memory means that the maximum number of words that can be generated and store is one per clock cycle. As showed in Figure 2.8, the first 4 words are identical to the cipher key.

This hardware generates the expanded key. The total size of this key is 1408 bits and can be calculated previous to the ciphering stage and then stored in memory. This represents an important area of storage, either as registers or a memory implementation, requiring a total of 44 words of 32 bits. Moreover a single-port memory would constrain the throughput of this operation due to its recursiveness and thus, the usage of a memory means that the maximum number of words that can be generated and store is one per clock cycle.

In order to reduce the area of the implementation, this work implements the key expansion as an on-the-fly calculation. As showed in Figure 2.8, the first 4 words are identical to the cipher key. The sequence to generate the next expanded key word includes the byte shift (*RotWord()* function application) of the previous word, w_3 or later $w_{(i-1)\times 4+3}$, and the transformation *SubWord()*. In order to optimize the area, only one S-Box S_R is implemented. The *SubWord()* hardware takes 4 clock cycles as it applies the substitution box on each of the 4 bytes composing the word. The *WordRcon* is selected with the iteration counter used as an index. This sequence is done in 4 cycles. At this point, each generation of the next 4 expanded key words use the previous word and the generated word 4 places behind. The previous word is temporary stored in a register so only the word $w_{(i-1)\times 4}$ needs to be read. The result of the XOR operation is written as a new word into the RAM. These memory accesses, one read and one write for each expanded key word, takes 2 cycles to be realized. The total time in cycle to process the full expanded key is given by the next equation:

$$\begin{aligned}
T_{EXPAND\_KEY} &= T_{write\ key} + (N_{words} \times T_{generate\ word} + T_{SubWord}) \times N_{iterations} \\
&= 4 + (4 \times 2 + 4) \times 10 \\
&= 124\ cycles
\end{aligned}$$

Data States

The two data states, *RECEIVE_DATA* and *SEND_DATA*, handle the channel handshake signals for receiving and sending data, respectively. As the AES cipher works on 128 bits, 4 words of 32 bits, it takes at least 4 cycles to receive or send all the words, depending of the availability of the external block. The words are stored and organized as byte in a data state as defined in Figure 2.2.

$$\begin{aligned}
T_{RECEIVE\_DATA} &= N_{words} \\
&= 4\ cycles
\end{aligned}
\qquad
\begin{aligned}
T_{SEND\_DATA} &= N_{words} \\
&= 4\ cycles
\end{aligned}$$

SUBBYTES

This FSM state enables the substitution of the bytes using the S-Box S_R. To limit the resource utilization, only one S-box is implemented as a ROM. It is the same resource as the substitution box used by the key expansion. As a data state is composed of 16 bytes, it takes 16 cycles to realize the full substitution process.

$$\begin{aligned}
T_{SUBBYTES} &= N_{bytes} \times T_{Sbox\ S_R} \\
&= 16 \times 1 \\
&= 16\ cycles
\end{aligned}$$

SHIFTROWS

This operation reorganizes the bytes of the state using a byte shifting of the rows. As the shift operation has a low cost hardware, all the required shifts are done in one cycle.

$$T_{SHIFTROWS} = 1\ cycle$$

MIXCOLUMNS

This state enables a matrix multiplication in GF(2^8) with a constant matrix. The byte multiplier uses the GF operator $Mul(x, y)$ defined in Section 3.1.1.

$$\begin{aligned} T_{MIXCOLUMNS} &= N_{words} \\ &= 4\ cycles \end{aligned}$$

ADDROUNDKEY

The *ADDROUNDKEY* state enables an XOR operation between each word of the data state and an expanded key word. The latter being stored in a RAM, one operation is done every cycle for a total of 4 cycles to process the 4 words data state.

$$\begin{aligned} T_{ADDROUNDKEY} &= N_{words} \times T_{RAM\ read} \\ &= 4 \times 1 \\ &= 4\ cycles \end{aligned}$$

The total number of cycles to realize an AES ciphering is given by the following equation:

$$\begin{aligned} T_{cipher} = {} & T_{RECEIVE\_DATA} + T_{SEND\_DATA} + T_{EXPAND\_KEY} + T_{ADDROUNDKEY} \\ & + (T_{SUBBYTES} + T_{SHIFTROWS} + T_{MIXCOLUMNS} + T_{ADDROUNDKEY}) \times 9 \\ & + T_{SUBBYTES} + T_{SHIFTROWS} + T_{ADDROUNDKEY} \\ = {} & 4 + 4 + 124 + 4 + (16 + 1 + 4 + 4) \times 9 + 16 + 1 + 4 \\ = {} & 382\ cycles \end{aligned}$$

This result supposes that the channels to receive and send data are always ready so it takes the minimum number of cycles to transfer the data state.

4.3.3 ZUC Cipher Block

ZUC cipher keystream generation block is described in this section. The operation of this block is really similar to the one described for SNOW 3G ciphering and thus we will mainly focus on the dissimilarities between SNOW 3G and ZUC ciphering.
The FSM operation described in Figure 4.6 applies to ZUC ciphering. The major structural difference in the LFSR is a 31-bit shift register of length 16. Also, a bit reorganization module is implemented.
The implementation of the FSM is done in conformance with the results obtained in Section 3, i.e. S-box S_0 is implemented on-the-fly since the area in this implementation is 58 % lower. As for the S-box S_0, the implementation is really similar to S-box S_R, involving inversion in GF and an affine transformation. Therefore, this S-box is implemented as a ROM. The number of substitution boxes used in the FSM is kept as low as possible, and consequently only one S_0 and one S_1 S-box is implemented. This leads to a FSM that generates 1 output every 4 clock cycles.

ZUC Feedback Implementation

As we saw in Section 2.1.4 the feedback path performs the function:

$$v = \left[2^{15}s_{15} + 2^{17}s_{13} + 2^{21}s_{10} + 2^{20}s_4 + (1+2^8)s_0\right] \; mod \; (2^{31}-1)$$

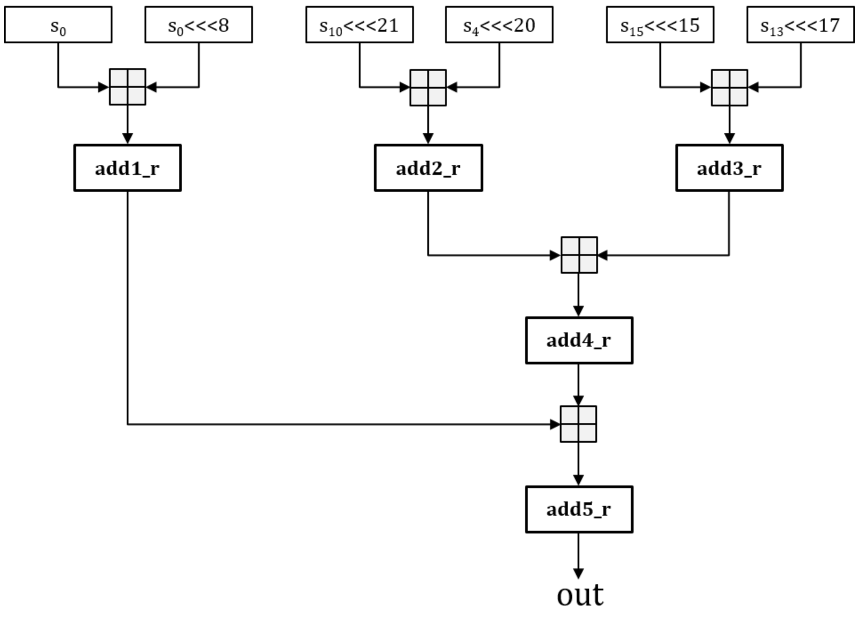
This function has been implemented in different ways. The first approach was to implement the feedback path in an adder tree configuration, while the second took advantage of the fact that we have 4 clock cycles to compute the feedback result. The two implementations are shown in figure Figure 4.8.

Adder Tree Implementation First, taking advantage of using an HLS tool we described the feedback path using a modulo operation % after every addition. This led to a high area figure in the feedback path, being this area close to 100 kilo-Gate Equivalent (kGE). During the second approach, the modulo was described as follows:

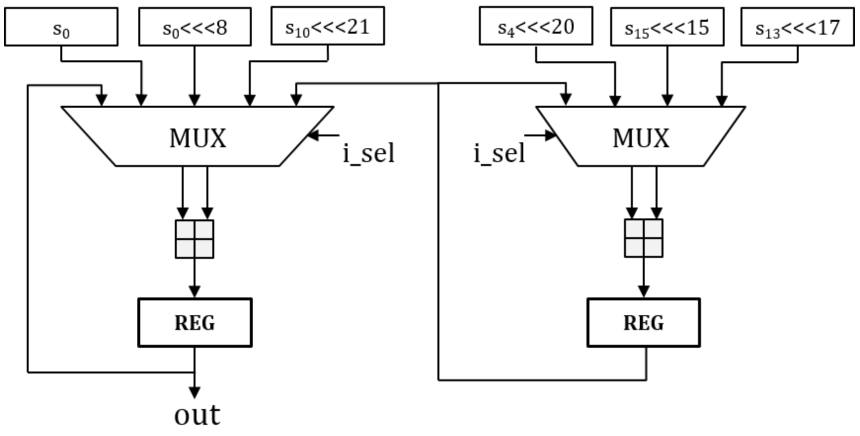
Proposed implementation 1:

```
const $TWO\_POW\_31\_M1 \leftarrow 0x7FFFFFFF$
$v = a + b$
if $v \geq TWO\_POW\_31\_M1$ then
    $v = v - TWO\_POW\_31\_M1;$
end
```

This implementation shows much better results than using a direct *C*++ modulo



(a) Adder tree



(b) Time multiplexed

FIGURE 4.8: ZUC feedback implementation

operation and letting Catapult implement it. The last implementation is taken from [12] section 3.2, where the following implementation is proposed:

Proposed implementation 2:

```
$v = a + b$
**if** *carry bit* **is** *1* **then**
    $v = v + 1$;
**end**
```

The results from the three implementations described are shown in Table 4.2. We observe how Catapult generates 5 adders for the adder tree plus one additional adder to perform each of the $v = v + 1$ operations on the second proposed implementation. In the first implementation proposed, the number of adders implemented by Catapult is increased to 15.

Time Multiplexed Implementation Four clock cycles are available to compute the feedback and so, a time multiplexed solution has been explored using Catapult Synthesis. This solution proves to be the most adequate one. For this implementation, we have based implemented the *modulo* based on the second proposed implementation. An architecture with two adders with multiplexed inputs and two registers has been implemented, leading to the results in Table 4.2. Also in this case, we observe two additional adders due to the $v = v + 1$.

TABLE 4.2: ZUC feedback adder tree vs. time multiplexed results

Implementation	Area (normalized)	Throughput (cycles)	Slack @ 50 MHz (ns)	N adders
Built-in modulo (%)	29.4	1	1.04	N.A.
Proposed 1	2.6	1	15.82	15
Proposed 2	2.1	1	16.41	10
Time multiplexed	1.0	4	16.08	4

4.3.4 Stream Cipher Block

It is noticed from the previous sections how the two stream cipher algorithms implemented are very similar in structure. From sections 3.5, 4.3.1 and 4.3.3 we can extract the following conclusions:

- The control logic used in SNOW 3G and ZUC ciphering as well as the handshake protocol with external modules is the same.

- The feedback loop is based on different arithmetic operations and therefore differs from one algorithm to the other.
- Both use a shift register of length 16. If a 32-bit shift register is used, it can be implemented by both algorithms.
- The FSM although presenting common operations, has been proven in Section 3.5 to provide better results if kept as independent modules.

With these results presented, SNOW 3G and ZUC stream cipher algorithms have been partially combined into a single block with common control and a common 32-bit shift register. The feedback and FSM modules have been kept independent. Some extra logic is necessary to be able to implement both algorithms in the same module. An algorithm select input is needed to select between SNOW 3G and ZUC keystream generation. This signal will be the *i_algo_sel* already mentioned in Section 4.3. Furthermore, the necessary multiplexing logic to connect the appropriate signals to the output and shift register input has been implemented. Table 4.3 summarizes the results obtained.

TABLE 4.3: Stream cipher algorithms combined

Implementation	**Area (normalized)**	**Throughput (cycles)**	**Slack @ 50 MHz (ns)**
SNOW 3G	1.3	4	15.08
ZUC	1.0	4	13.80
Total	**2.3**	**-**	**-**
Combined	**1.8**	**4**	**12.95**

The combination of SNOW 3G and ZUC leads to an optimized design exhibiting 22% less area footprint than using 2 independent cipher cores. Thus, the final implementation of the cryptographic processor will use this combined stream cipher module.

4.4 Confidentiality Block

The following section details how the confidentiality algorithms defined in Section 2.2 have been implemented. The confidentiality block is presented in Figure 4.9.

This block can be started only when the signal *i_integrity_en* is '0', and will perform one of the EEA depending on the *i_algo_sel* input. This module contains

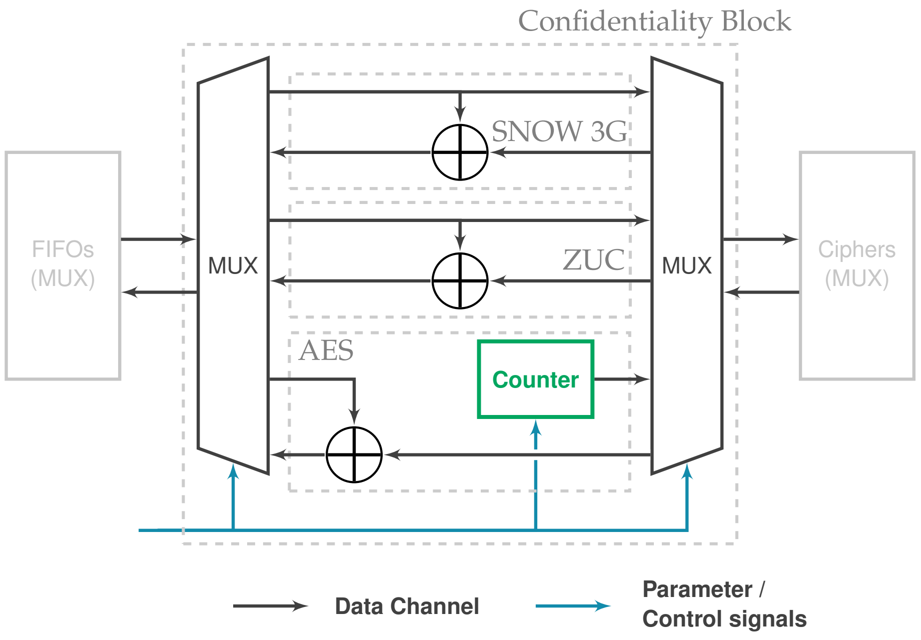


FIGURE 4.9: Confidentiality block diagram

the logic of each confidentiality algorithm, multiplexed with the algorithm select. Four channels are present as interface for this component. One input and one output channel are connected respectively to the TX and RX FIFO through the channel multiplexer. Another output and input channel are part of this block and connected to the cipher core through the second channel multiplexer. The confidentiality block handles these four channels in order to realize the EEA functionality by using the cipher processing element.

For SNOW 3G and ZUC, both stream cipher, the block will perform an XOR operation between the keystream coming from the cipher block and the input data from the TX FIFO. The handshake signals between the FIFOs and the cipher block is bypassed by this block as the cipher module handles these signals for SNOW 3G and ZUC. The logic is kept minimal to process the confidentiality operations.

On the other hand, in the case of AES, the cipher block must be loaded with a set of counters T_i that represent the inputs of the AES cipher block in CTR mode. Therefore, this block will generate the necessary handshake signals towards the input/output FIFOs and towards the cipher block as defined in Section 2.2. The AES logic of the module must create and provide the data to the AES core in order to obtain the correct output used as a keystream. The keystream output originated from each of the input counters to the cipher block will then be XORed with the input message. The confidentiality AES uses the context parameters provided by the AHB IF controller and set a counter incremented for each block of data sent to the cipher core. When receiving the ciphered data, it synchronizes the keystream words with the input data received from the TX FIFO in order to XOR the data word to produce the ciphered/plain text for encryption/decryption. This synchronization also manages the discarding of words produced by the cipher block. As detailed in Section 2.1.2, the AES cipher core always produces a block of data, i.e. 4 words of 32 bits. The plain/ciphered text to process may not have a length multiple of 4 words. For the last part of text when the number of text words is less than 4, all of the 4 produced words by the cipher can not be used. They are acknowledged by the confidentiality logic anyway in order to empty the AES cipher core of them.

4.5 Integrity Block

The integrity block contains the hardware logic to implement the integrity algorithms defined in Section 2.3. This section presents the integrity module, illustrated in Figure 4.10, part of the security IP.

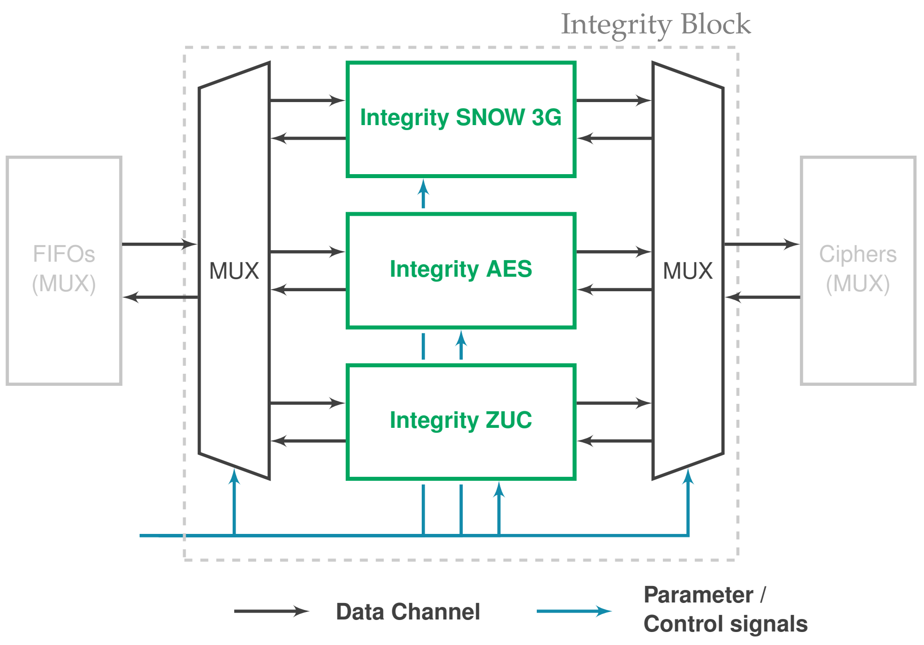


FIGURE 4.10: Integrity block diagram

As for the confidentiality block, introduced in Section 4.4, the integrity module has four channel interfaces: two inputs from the TX FIFO and the cipher core through channel multiplexers; and two output channels that connect to the RX

FIFO and the cipher core input through channel multiplexers. The module is enabled when the signal *i_integrity_en* is high. It will perform one of the EIA after receiving '1' on its start signal *i_start* and depending on the *i_algo_sel* input. The module multiplexes the three integrity algorithm's hardware logic. The following sections detail the implementation of integrity protection for each of the SNOW 3G, AES and ZUC algorithms. In each of the FSMs presented, every state with a name starting with *SEND* or *RECEIVE* are communication states. They include the handling of channel handshake signals.

4.5.1 SNOW 3G Integrity Block

This integrity block implements the SNOW 3G integrity algorithm defined in Section 2.3.1. The SNOW 3G cipher is used to produced 5 keystream words. During the first round of the FSM illustrated in Figure 4.11, in the state *RECEIVE_-FROM_CIPHER*, two keystream words are acquired from the cipher and stored.

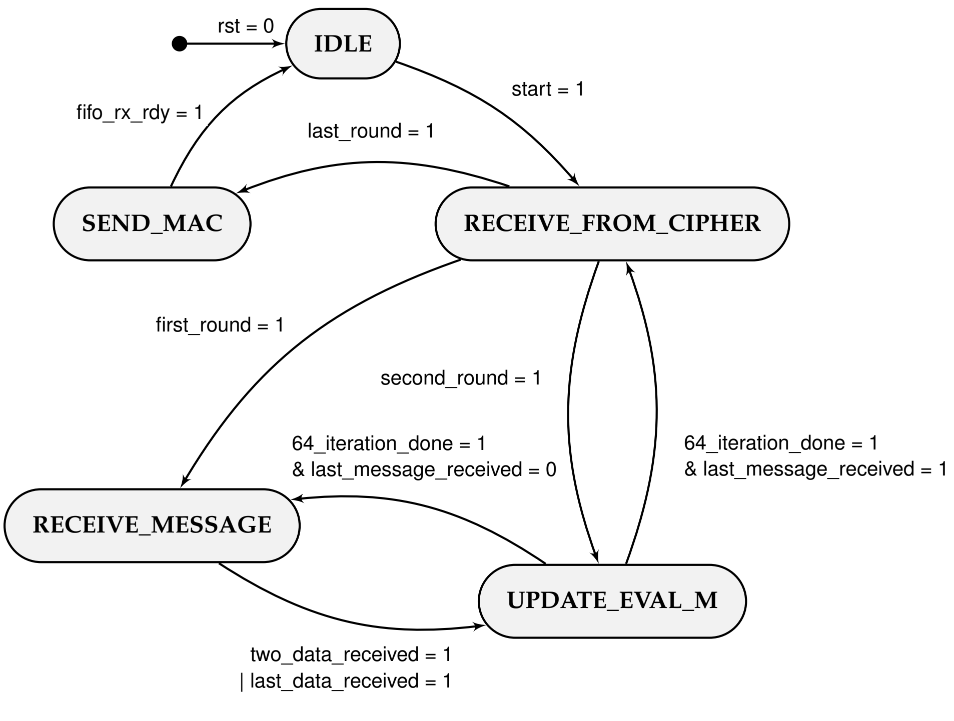


FIGURE 4.11: FSM of SNOW 3G integrity block

The following state is *RECEIVE_MESSAGE* where two message words are received and stored into registers. The main processing is realized in *UPDATE_-EVAL_M*. The previously saved values are used to compute the update of the 64 bits register *EVAL*, which takes 64 iterations. The detailed hardware logic of this operation is illustrated in Figure 4.12. All the operations are realized on 64 bits. Two new message data words are received and used in the following *UPDATE_EVAL_M*. This process is repeated until all the message words have been consumed. At the end of the last *UPDATE_EVAL_M* processing the last message data, two keystream words are taken from the cipher. During this second round, the state *UPDATE_EVAL_M* realized the same operation to update *EVAL* but with these new keystream words and the length of data as input words instead of message data.

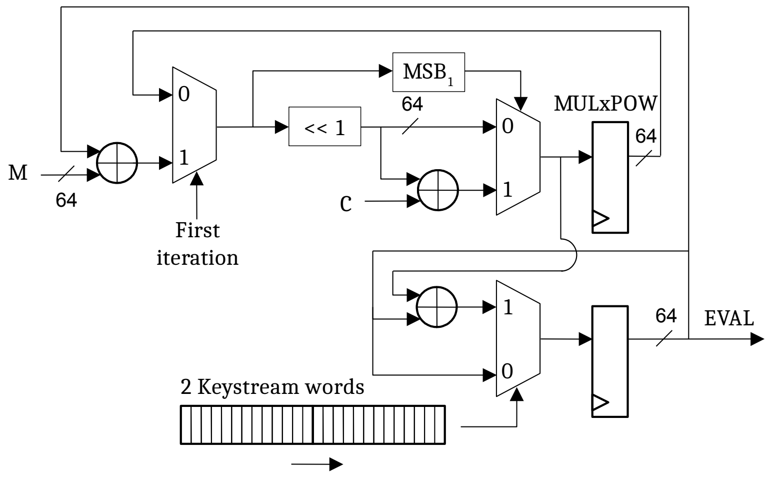


FIGURE 4.12: Diagram of *EVAL_M*

Finally a last keystream word is obtained and XORed with the most significant word of the final value of *EVAL*. This result is the 32 bits word *MAC* that is sent in the last state *SEND_MAC*.

4.5.2 AES Integrity Block

The AES integrity algorithm is presented in Section 2.3.2. This module implements the hardware to realize the integrity features of AES in the CMAC mode. This module handles the handshake signals of the integrity channel interfaces, computes the subkeys $K1$ and $K2$, provides the message data to the AES cipher engine, and extracts the MAC word from the cipher results. These operations are sequenced with the FSM illustrated in Figure 4.13.

After receiving a start signal, the FSM state goes to *SEND_TO_CIPHER*. As its name shows, this state sets the data to be used by the cipher core and sends it. It always last at least 4 cycles as a block of data to sent to the cipher is composed of 4 data words. On its first call, the data is set to zero. The first ciphering is used to produce the ciphered text to generate the subkeys $K1$ and $K2$. On the other call, the data is either the received message data, the message data XORed

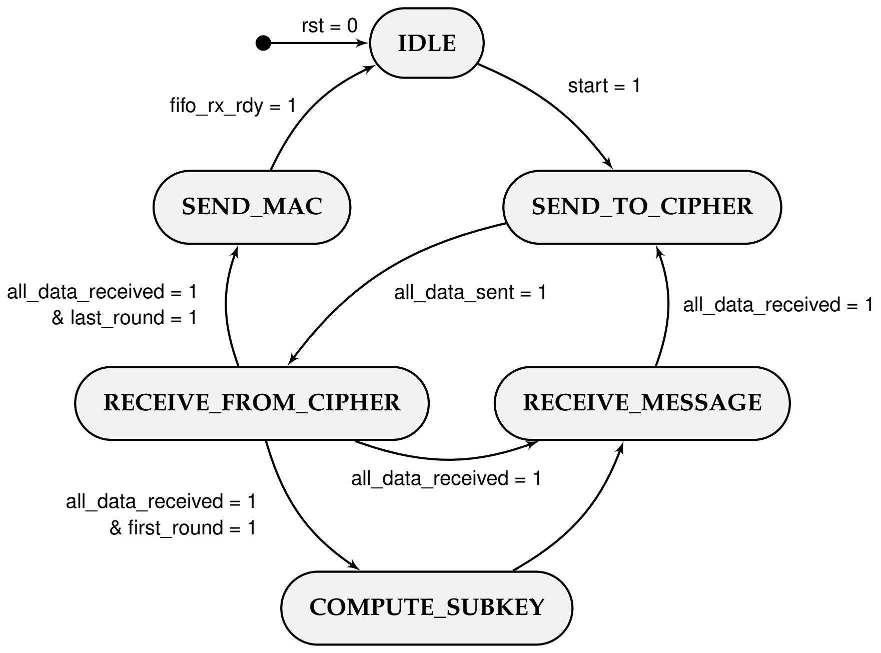


FIGURE 4.13: FSM of AES integrity block

with the cipher output, and for the last call also XORed with the subkey. This is illustrated in Figure 2.14. Once all the 4 words have been sent to the cipher through the channel and have been acknowledged, the next state is *RECEIVE_-FROM_CIPHER*. This state will handle the data received from the cipher core by controlling the channel and its handshake signals. After the 4 words resulting from the ciphering have been acknowledged, the FSM changes its state to either *COMPUTE_SUBKEY* when it is the first call, *SEND_MAC* when it is the last call, or *RECEIVE_MESSAGE* otherwise. *COMPUTE_SUBKEY* state is reached a single time and lasts one cycle. It computes the subkey $K1$ and, only when necessary, $K2$ as defined in the algorithm 2.2. The FSM state *RECEIVE_MESSAGE* allows the module to receive the message data from the TX FIFO. Most of the time, 4 words are obtained from the FIFO but for the first block of data, only two words are expected. As specified in Section 2.3.2, the first data block is composed of two words from the context and two words from the message. The context data is directly available to the module as it is passed as a parameter from the registers present in the AHB IF controller. The last time the state *RECEIVE_MESSAGE* is the current state, the number of data to receive is between 1 and 4 depending on the data length in word modulo 4 words. The last state is *SEND_MAC* and handles sending of the resulting MAC. This 32 bits word is taken from the output of the last cipher processing.

4.5.3 ZUC Integrity Block

This block implements the algorithm presented in Algorithm 2.3 in Section 2.3.3. The following FSM, Figure 4.14 is implemented in order to compute the 32 bits word MAC. To summarize, the algorithm recursively XORs words from the generated keystream when the bit at the iteration index has a value of '1'. To achieve this, the first state after the module receives an active start signal is *RECEIVE_-MESSAGE*. One message word is stored in a register and the FSM current state is set to the next state, *RECEIVE_FROM_CIPHER*. The first time this state is reached, two keystream words from ZUC cipher are obtained, while only one is saved the subsequent times. The following state is *UPDATE_T*. The 32 bits register *T* contains the result of all the recursive XOR iterations.

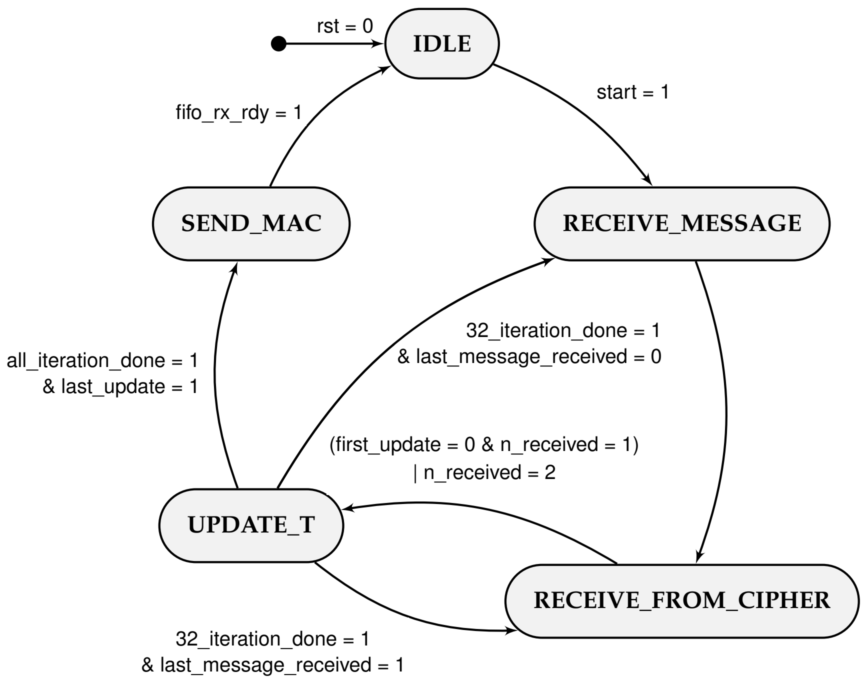


FIGURE 4.14: FSM of ZUC integrity block

As showed in Figure 4.15, the two keystream words are concatenated. An index pointer *i* selects 32 bits out of the keystream and a bit from the message. If the message bit $M[i]$ is '1', the register *T* is updated with its previous value XORed with the pointed keystream word. This computation is repeated until *i* reaches 32. It means the first 32 bits of the concatenated keystream word has been consumed, as the full message word. The FSM state moves back to *RECEIVE_MESSAGE* first, then to *RECEIVE_FROM_CIPHER*. With a new message word *M* and a new keystream word, the process is repeated. Once the last message word has been received and used, *T* is XORed two more times with keystream words, without condition. The last value of *T* is the MAC that will be send in the next state *SEND_MAC*.

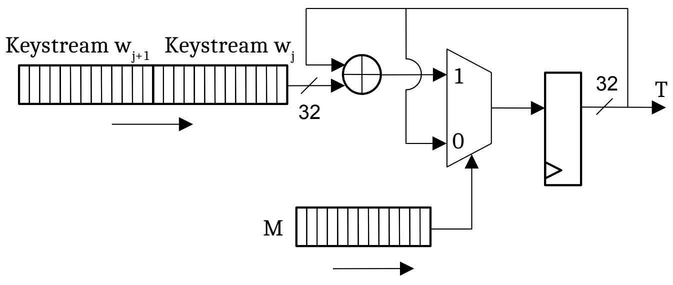


FIGURE 4.15: Diagram of *UPDATE_T*

4.6 Results

In this section, the synthesis results from Catapult are presented. Each module has be run through the tool in order to obtain the RTL source. The sample library for 65 nm technology is used. The area value provided by Catapult was converted to an equivalent number of gates. A NAND gate has been synthesized and the resulting area has been used as the conversion factor to obtain the number of gates from the Catapult area:

$$Area_{NAND} = Catapult\ area\ (65nm)\ /\ 1.6$$

The area results of the IP and its internal modules are reported in Table 4.4 normalized against the smallest design, corresponding to ZUC integrity block.

TABLE 4.4: Catapult synthesis results

Module	Area (normalized)	Area Ratio (%)
Cipher: Combined SNOW 3G and ZUC	6.4	32.0
Cipher: AES	3.5	17.7
Integrity: SNOW 3G	1.9	9.5
Integrity: AES	3.0	15.2
Integrity: ZUC	1.0	5.1
Controller	1.8	9.2
Others (Confidentiality, MUXes, . . .)	2.3	11.3
Security IP	19.9	100.0

Chapter 5

Verification and Benchmarking

Verification has been done at different hierarchy levels, ranging from individual cipher and integrity blocks to simulations of the crypto processor as a whole. Furthermore, this verification has also been done at different abstraction levels. Starting from C++ simulations, the ciphering and integrity blocks have been tested at a functional level. Then, AES, SNOW 3G and ZUC ciphering and integrity blocks have been run through Catapult Synthesis to generate the RTL description of these modules, which has then been verified at a behavioral level using an RTL testbench.

The control, data path and confidentiality blocks have been described using System C language. These modules has been directly verified using cycle accurate simulations. Finally, to test the crypto processor, cycle accurate simulations have been run on a system including an ARM processor, and the crypto processor connected through an AHB bus.

5.1 C++ Verification

The main hardware processing elements such as the ciphering and integrity block have been developed using HLS C++ with Algorithmic C (AC) libraries. The source code is interpreted by Catapult in order to generate RTL modules. Before this step, the functionality of the block is tested using a software testbench written in C++. The scope of this verification is to control the compliance of the described functions to the 3GPP specifications. The focus is not placed on the hardware particularity such as the channel interfaces or the complete flow. This will be the scope of the other levels of control and presented in the next sections, RTL Verification and Cycle Accurate Simulations. Each block is individually verified with a C++ testbench containing the testcases data from the specifications.

A *run_test* function is defined to execute a single of the block tested. The arguments given to the function are the parameters (key, context, data length,. . .), the input data (either plain text or message) and the expected ciphered data. The test function provides the needed data to the block under test as a C structure. The block is run as a function call, which returns an output structure containing the output values. Using the returned value from this function, *run_test* selects the future data whom are set as the input structure.

As the block is described as a cycle accurate C++ function, this is executed inside a *while* loop. Algorithm 5.1 illustrates as pseudo-code the structure of the *run_test* function.

```
Let struct_i be a structure of all the input elements.
Let struct_o be a structure of all the output elements.
Set struct_i ← arguments as parameters.
// Start the block
struct_i.i_start ← 1;
Call block.run(struct_i, struct_o);
struct_i.i_start ← 0;
// Run each cycle until the block has finished
while struct_o.o_block_rdy ≠ 1 do
    // Run the block for 1 cycle
    Call block.run(struct_i, struct_o)
    // Set input data when block is ready
    if struct_o.o_fifo_in_ack = 1 then
        struct_i.i_data_vld ← 1;
        struct_i.i_data ← input_data[cnt_in++];
    else
        struct_i.i_data_vld ← 0;
    end
    // Acknowledge and check the output data
    if struct_o.o_data_vld = 1 then
        struct_i.i_fifo_out_ack ← 1;
        if struct_o.o_data = ref_data[cnt_out++] then
            error_cnt++;
        end
    else
        struct_i.i_fifo_out_ack ← 0;
    end
end
Display error_cnt
```

Algorithm 5.1: Pseudo-code of the testbench function *run_test*

The three sections inside the *while* loop, function call, input data and output data, can be interchanged. Even though C++ is a sequential language, the usage of handshake signals allows a parallelism of the input, output and block section during the simulation. The *run_test* function is used with each test data set allowing to verify the functionality of the module independently of the previous run.

5.2 RTL Verification

From the HLS C++ code, Catapult generated RTL sources. The modules are verified at this level of description in order to verify the functionality and particularly the channel handshake interfaces. This is realized with RTL testbenches that have been written for the cipher and integrity blocks. The module to test, also named Unit Under Test (UUT), is instantiated in the testbench along with stimuli generator and result checker. The input data channel of the cipher is connected to the stimuli generator. This source provides the module with the testcase's data respecting the channel signaling. It also defines various delays for the valid data in order to fully simulate the handshake protocol. This can be observed in Figure 5.1. In this simulation, the signal *tb_i_data_vld*, part of "Channel in" group, is defined in the testbench to represent a rarefaction of the input data. For the first five data, the cipher is producing data at maximum throughput as the delay between two input data valid is less than 4, the throughput of the cipher SNOW 3G. As the input data rate continues to slow down, the cipher equally reduce its generation of keystream words to stay synchronized. It can be noticed that the output signals *tb_o_data_vld* and *tb_o_fifo_in_ack* always match each other.

Figure 5.2 represents a full SNOW 3G cipher process is simulated to generate two keystream data words. The one cycle active start, *tb_i_start*, launches the hardware ciphering. The signal *tb_o_cipher_rdy* indicating that the module is idle is de-asserted. SNOW 3G cipher processes through the initialization mode describes in Section 2.1.1. It lasts for 32 iterations of 4 clock cycles. Directly following this sequence, the first keystream data produced is discarded. It can be seen in the figure, the signal *tb_o_z* is set to a value after the initialization mode without an active valid *tb_o_data_vld*. The first valid keystream data is generated $T_{first\ keystream}$ cycles after the start signal, defined by the following equation:

$$\begin{aligned} T_{first\ keystream} &= T_{initilization} + T_{discard} + T_{data} \\ &= (N_{init\ iteration} + N_{discard} + N_{data}) \times N_{thoughput} \\ &= (32 + 1 + 1) \times 4 \\ &= 136\ cycles \end{aligned}$$

This is visible in the figure by counting the number of transitions in the output data signal *tb_o_z* and multiplying this value by the throughput of the cipher, 4.

The testbench is self-checking, meaning that it contains the expected output data. Each data received from the UUT is compared with the reference value. An error message is triggered when a mismatch happens. Each simulation is concluded with the detection of the idle state of the tested module, realized with the *tb_o_-*

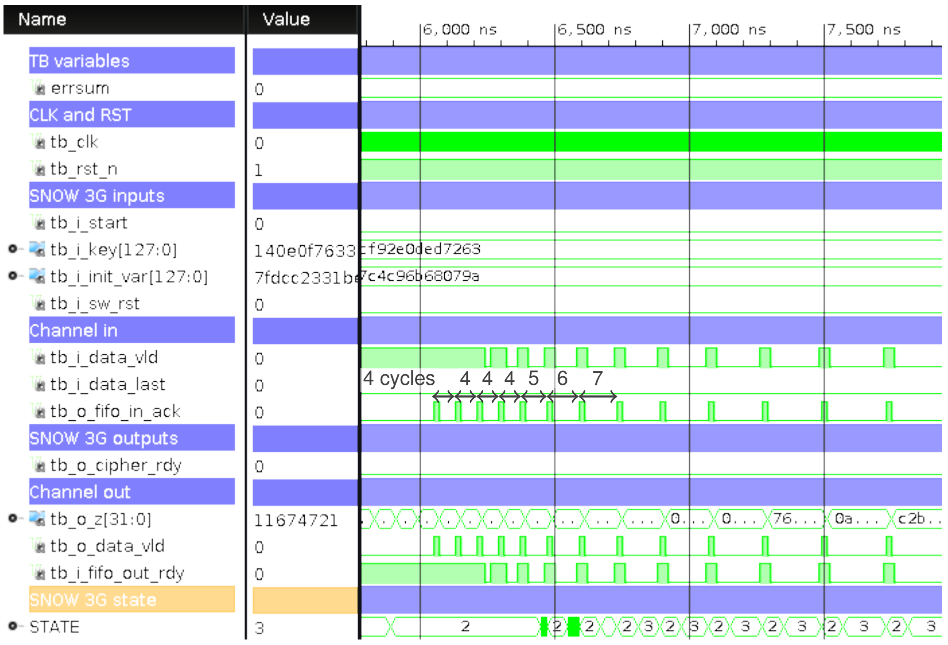


FIGURE 5.1: RTL simulation of the SNOW 3G cipher with various data delay

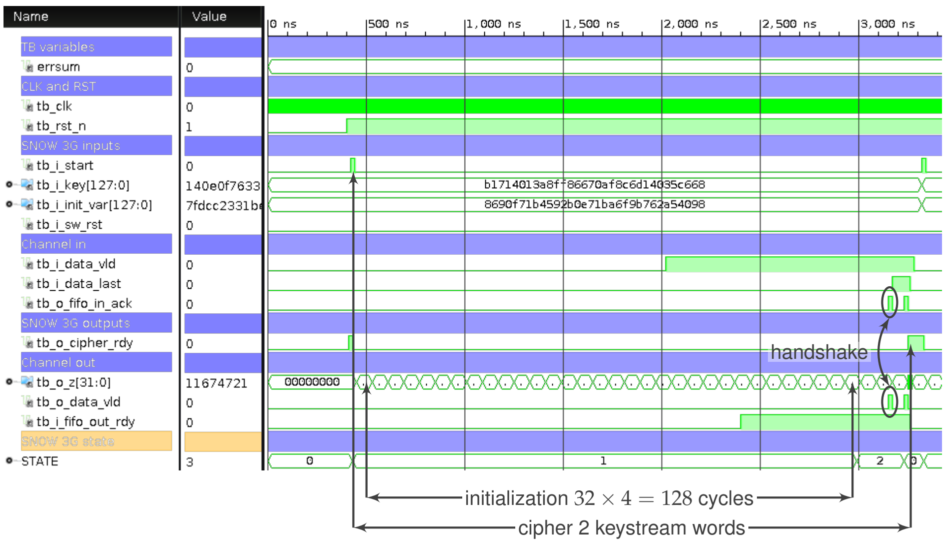


FIGURE 5.2: RTL simulation of the SNOW 3G cipher generating 2 keystream words

cipher_rdy after the last test set. A summary message is defined to list the passing and failing tests to overcome the need of visual control of the waves.

The simulation of a ciphering using AES algorithm is illustrated in Figure 5.3. The testbench used for this simulation is identical, except for the data that comes from AES testcases. As described in Section 4.3.2, the input and output data are not processed continuously with this block cipher. The first phase is the key expansion. It is followed with the request of 4 input data words. After the cipher run, a cipher state composed of 4 data words is provided as the result.

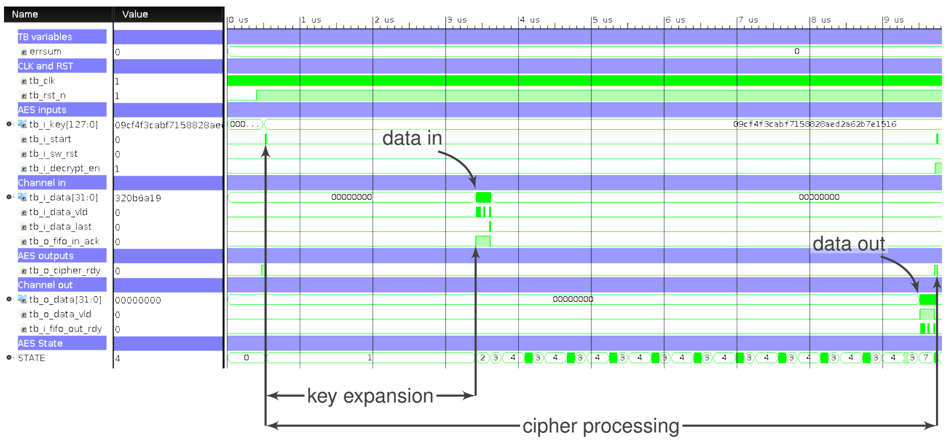


FIGURE 5.3: RTL simulation of the AES cipher

Figure 5.4 is an annotated waves image of AES integrity. We observe the different phases of the integrity block communicating with the cipher module as explained in Section 4.5. To process the AES integrity algorithm on a data state, i.e. 4 words, the cipher element is used two times: first time to cipher data of zeros in order to compute the subkey; and a second time to cipher the message data XORed with the subkey (as the message length is less or equal to 128 bits).

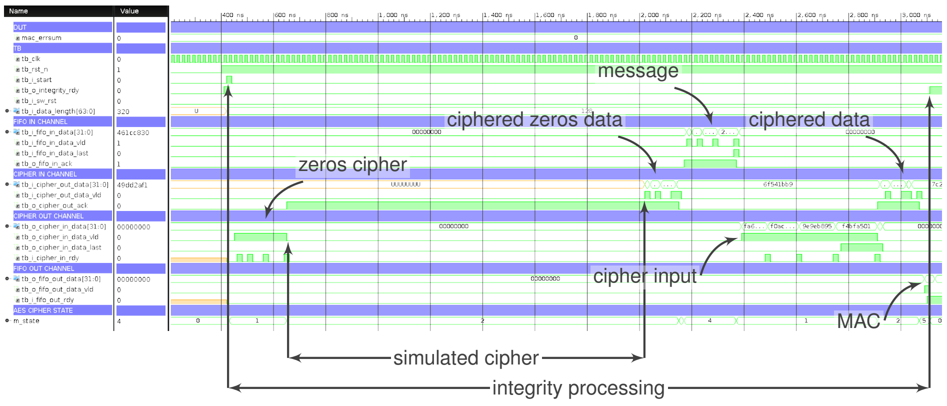


FIGURE 5.4: RTL simulation of the AES integrity

The Figure 5.5 illustrates the SNOW 3G integrity block. The first two keystream z_1 and z_2 are used with the message data, employed two by two, in order to update the *EVAL* value. The next two keystreams, z_3 and z_4, are used with the message length available as a parameter input to the module. Finally the result is XORed with the last keystream z_5 to produce the MAC word.

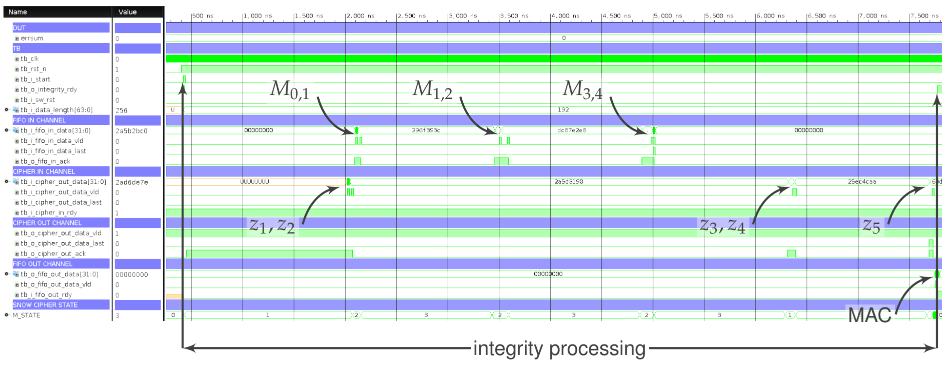


FIGURE 5.5: RTL simulation of the SNOW 3G integrity

The last wave caption in Figure 5.6, represents two runs of ZUC integrity module. This block always uses two keystream words more than the number of message words. Two keystream words, z_1 and z_2, are always requested first after the first messsage data.

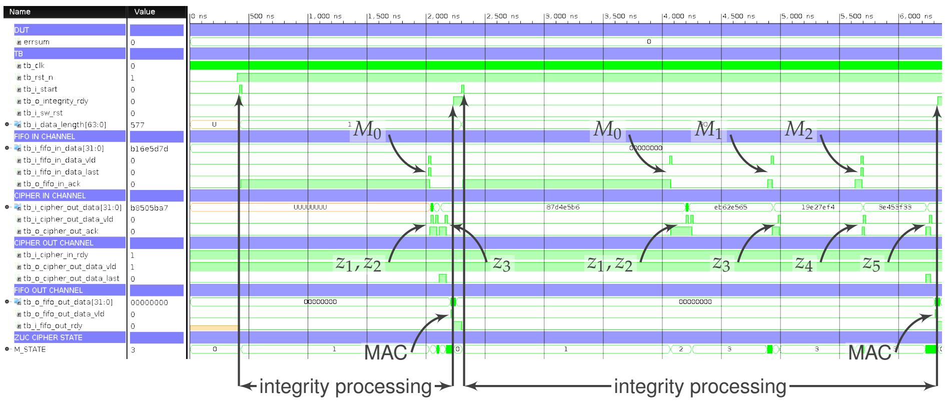


FIGURE 5.6: RTL simulation of the ZUC integrity

It must be noticed that the timing represented in some of the waves do not reflect a realistic latency and time scenario. The cipher modules and the FIFOs are modeled in the testbench with different timing. This is done in order to test the handshake signals of the UUT's channels.

5.3 Cycle Accurate Simulations

SoC Designer cycle accurate simulations allow the designer to easily perform architectural exploration, system analysis and software bring-up, permitting early stage virtual prototyping of a System on Chip (SoC). The technology of such tool relies on RTL compilation which will guarantee a perfect match between the hardware subject to virtualization and the RTL. This tool will thus ensure functional and cycle accuracy. Previous works [10][11] have used this same approach to evaluate system performance.

The environment has been set up as shown in Figure 5.7. The crypto processor is shown in green on the right hand side of the figure. The hardware components of test environment have been classified in the following manner: in blue we can see the processor subsystem and all the blocks related to the processor (memories, IOs, ...). In yellow we observe the AHB bus related components such as the AHB matrix and some interconnect modules which have no actual hardware relevance, since they are used to perform software conversion between types.

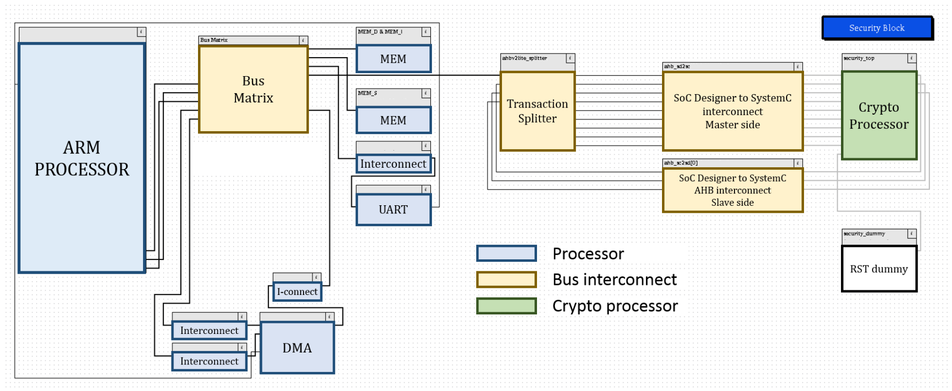


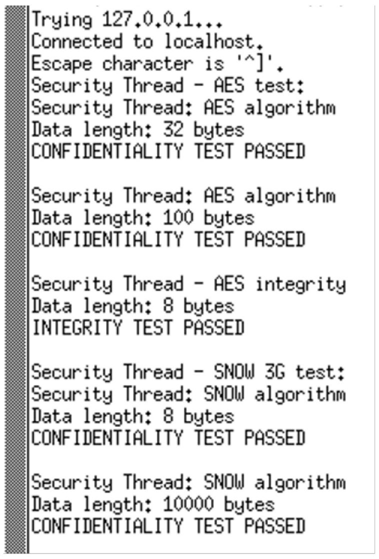
FIGURE 5.7: Test environment on SoC Designer

Using this set up, a simple software driver and a testbench has been developed. The core functions of the driver allow to send and receive data through the AHB bus. Then, some other routines have been developed to:

- Get the content of the version register in the IP.
- Set/get the content of the Control/Status registers. Note that writing a RO register will have no effect.
- Set/get the content of the key and context registers in the IP.
- Set/get the content of the data length registers in the IP.
- Send plaintext and receive ciphertext data to-from the FIFOs in confidentiality mode. Send data and receive a *MAC* word in integrity mode.

Refer to figure Table 4.1 for the list of software accessible registers in the IP. These software functions have been used to perform different confidentiality and integrity test cases for each algorithm. These test cases are a subset of the available ones for each algorithm at the time this report has been written. The test vectors to be provided to the crypto processor can be found in [6] for AES, [9] for SNOW 3G and [13] for ZUC.

Figure 5.8 shows the output prompted in a hyperterminal showing the results of the testcases provided.



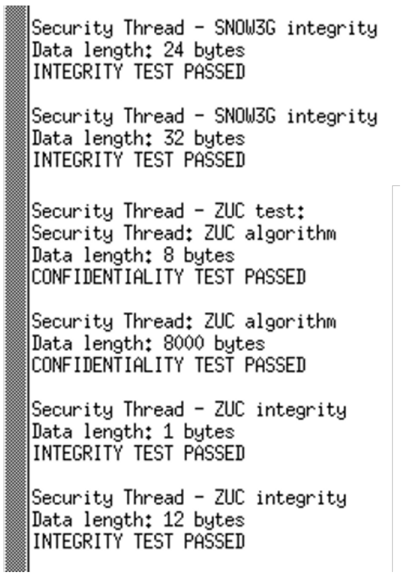
```
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
Security Thread - AES test:
Security Thread: AES algorithm
Data length: 32 bytes
CONFIDENTIALITY TEST PASSED

Security Thread: AES algorithm
Data length: 100 bytes
CONFIDENTIALITY TEST PASSED

Security Thread - AES integrity
Data length: 8 bytes
INTEGRITY TEST PASSED

Security Thread - SNOW 3G test:
Security Thread: SNOW algorithm
Data length: 8 bytes
CONFIDENTIALITY TEST PASSED

Security Thread: SNOW algorithm
Data length: 10000 bytes
CONFIDENTIALITY TEST PASSED
```



```
Security Thread - SNOW3G integrity
Data length: 24 bytes
INTEGRITY TEST PASSED

Security Thread - SNOW3G integrity
Data length: 32 bytes
INTEGRITY TEST PASSED

Security Thread - ZUC test:
Security Thread: ZUC algorithm
Data length: 8 bytes
CONFIDENTIALITY TEST PASSED

Security Thread: ZUC algorithm
Data length: 8000 bytes
CONFIDENTIALITY TEST PASSED

Security Thread - ZUC integrity
Data length: 1 bytes
INTEGRITY TEST PASSED

Security Thread - ZUC integrity
Data length: 12 bytes
INTEGRITY TEST PASSED
```

FIGURE 5.8: Succesful test cases results on SoC Designer

The crypto processor must be wrapped to be able to import it in the SoC Designer environment. This is a fairly easy process which will convert the System C interface into the type of interface that SoC Designer uses to perform the simulations.

The debugging capabilities have been outstanding by using this methodology. Apart from the possibility of interfacing the block from a processor subsystem, and developing/testing a software driver during hardware implementation, two other debug capabilities are offered:

1. Using the system C tracing library has allowed us to see the internal signals in our design. The trace file is created in the SoC Designer wrapper module and will store all the activity which is dumped in a *.vcd* file. Then, a waveform viewer can be used to open this file and debug the hardware block as you would do on a normal RTL simulation.
2. The tool offers the possibility to trace the AHB bus transactions, which has also been of using during debugging.

Figure 5.9 shows the aforementioned internal waveform viewer during a SNOW 3G confidentiality test. The blue marker represents the data cycle of the AHB transaction that writes a logic '1' of the *start_bit* of the *CONTROL/STATUS* register. From this moment, cycle 16.790, a SNOW 3G confidentiality is started.

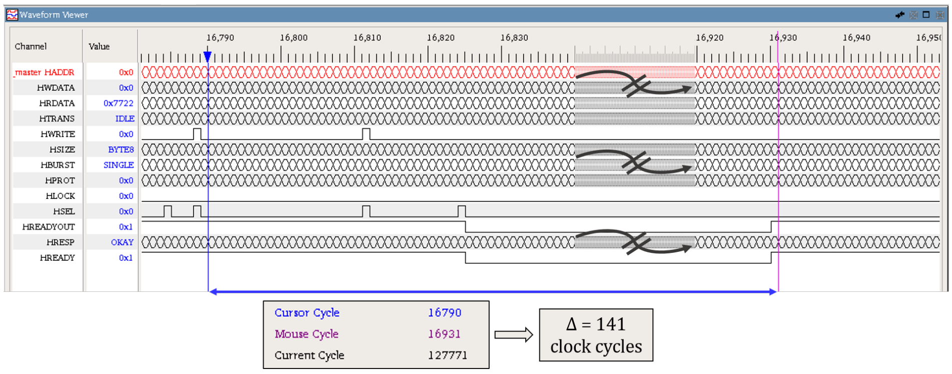


FIGURE 5.9: SNOW 3G AHB bus transactions

We observe the AHB master will be able to sample the first data coming from the crypto processor on clock cycle 16.931, which translates into a total of $\Delta = 141$ clock cycles in the communication. The delay is explained in Figure 5.10, where we observe that SNOW 3G latency is 136 clock cycles as we calculated in Section 4.3.1 and Section 5.2.

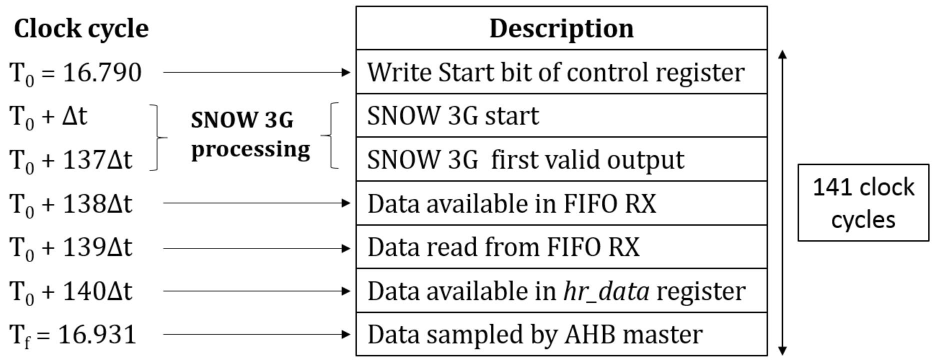


FIGURE 5.10: SNOW 3G initialization cycles

Chapter 6

Conclusions and Further Work

In this last chapter, we conclude our Thesis' work with a summary of the work developed. This chapter then follows with a comparison of our crypto processor with similar constructions. Finally, we will end the document putting forward some ideas of further work related to this project.

6.1 Conclusion

In this Master's Thesis we have successfully implemented a security processor capable of implementing the confidentiality and integrity algorithms specified by 3GPP for LTE. Using HLS has demonstrated to be very helpful for exploring different implementations before realizing the final design. For this matter, we have found that implementing an untimed C++ version of a module gives more flexibility when using Catapult Synthesis. Then, when an architecture is selected we found converting this untimed C++ implementation to a sample based implementation gave us more control on the generated RTL by Catapult. However, as we discussed in Section 1.4.2, by using this sample based approach it will be harder to perform architectural changes in the future if, for instance, a different technology or clock frequency in targeted.

From the first implementation to the final presented in this document, some blocks in the design have suffered major changes. Table 6.1 shows the impact of these changes on the design area.

SNOW 3G optimizations are due mainly to the change of a pipelined feedback to a time multiplexed implementation, as was described in Section 4.3.1. ZUC optimizations are also based on the feedback loop. In this case, the difference resides in how the modulo addition is performed in the feedback loop, as was discussed in Section 4.3.3. AES most straightforward optimization was to dispose

TABLE 6.1: Impact of optimizations on final design

Module	Area (normalized)	Area reduction (%)
SNOW 3G	1.6	19.7
SNOW 3G optimized	1.3	
ZUC	1.3	21.1
ZUC optimized	1.0	
SNOW 3G + ZUC	2.3	22.1
Stream Cipher	1.8	
AES	1.1	10.3
AES optimized	1.0	

of all hardware related to AES decryption, since CTR mode only uses the cipher block in encryption mode.

6.2 Comparison with State-of-the-Art

Firstly, most of the literature implementations target Field-Programmable Gate Array (FPGA) as it is a more accessible platform. The comparison between ASIC and FPGA area is not possible due to the differences of the area measurement: the kilo-gates equivalence (kGE) is used for ASIC while the area is reported as number of basic elements provided by the FPGA(registers, LUT, slices,...). No equivalence exists between them as a FPGA LUT can implement both a basic gate or a more complex combinatory logic. On ASIC, the area result would provide a quite different number of gates. Moreover it depends on the FPGA used as the properties of its elements may differ.

The second point is the difference in target technology and application scenario. When the platform used in the literature is an ASIC, the target is LTE and LTE-A high performance applications. In this Master's Thesis' work, we tried to achieve the lowest area at the cost of throughput, targeting a NB-IoT platform. A comparison with hardware accelerators aimed for low throughput technologies such as LTE Cat 1, LTE Cat M1 or NB-IoT would be more adequate.

Finally as a last comment, our area figures are provided by Catapult HLS. As previously described, this tool allowed to generated RTL from C++/SystemC. It does not provide optimized area results as other classical synthesis tools generate from RTL source code. For a 16 32-bit shift register, we reported a difference of 16% lower area when synthesized with Synopsys Design Compiler compared to Catapult HLS. More comparisons showed that the area difference may be between 10 to 40%, always at the disadvantage of Catapult HLS.

In Table 6.2, we have compared our crypto processor design with state-of-the-art designs.

TABLE 6.2: Comparison with state-of-the-art

Implementation	Area (normalized)	Data Rate (Mbps)	Technology (nm)
AES [32]	1.0	400	90
This work (AES)	1.7	100	65
Stream Cipher [42]	2.3	2700	90
Stream Cipher [29]	3.7	28800	–
Stream Cipher (this work)	3.1	1100	65
Crypto Processor [41]	5.3–20	–	–
Crypto Processor (this work)	9.5	–	65

Taking into account the previous comments, we found the closest matching existing work. For a complete Crypto Processor, very few existing works implementing all the 3GPP's security algorithms for LTE were found in the literature. A similar IP from Synopsys is available [41]. When we compare their design with our solution, we notice that our proposal is in the lower range of their Crypto Processor. However, they do not clearly define what is included in their smaller area result. The sources are not always clear whether the provided results include all the memories needed for the processing. The same can be said about the communication interface.
Our design results include all memories needed, both for the cipher processing and the data path (FIFO), as well as all interconnect interfaces, both internal with handshake channels and AHB IF to the external CPU.
In addition, we observe how the reduction in area comes at the expense of a reduction in the throughput of our IP in comparison with other designs.

Finally, more accurate and realistic area figures should be obtained from a RTL synthesis tool in order to increase the confidence on the results.

6.3 Further Work

Further work can be done on optimizing the key expansion for AES. The actual version will uses a memory to store the 44 words produced by this key expansion. An on-the-fly implementation can be implemented, which would remove the need for a memory and would also reduce the area and could reduce the power consumption of the design. The latter is not true if multiple blocks are encrypted using the same key [30].

In addition, the S-boxes implementation analysis can be expanded. In [16] a different S-box implementation is used in different parts of the design, depending of the usage given to each S-box. A power analysis tool would help in this further study to be able to take power the consumption of different implementations into account.

Besides AES, fine tuning of all the blocks implemented in HLS can be done. Catapult does a good job on optimizations, however we have seen how Catapult is not able to perform some bit level optimizations, like trimming down the width of registers if they are not defined at the precise bit width. For instance, if a state register for a state machine or a counter register is defined without care counting on HLS tool optimizations to correct for this.

The throughput of the design can be decreased since it is more than enough for the target technology: NB-IoT. For instance, the possibility of using 8 or 16-bit arithmetic and its impact on the design area can be studied. This could make the design more area efficient at the expense of performance.

Finally the design could be implemented and tested on FPGA and/or ASIC as a next step.

References

[1] 3GPP. *Series 33: Security aspects*. URL: http://www.3gpp.org/DynaReport/33-series.htm.

[2] 3GPP. *Series 35: Security algorithms*. URL: http://www.3gpp.org/DynaReport/35-series.htm.

[3] 3GPP. *Specification Numbering*. URL: www.3gpp.org/specifications/79-specification-numbering.

[4] 3GPP. *Technical Report TR 35.924: Document 4: Design and Evaluation Report*. URL: www.3gpp.org/DynaReport/35924.htm.

[5] 3GPP. *Technical Report TS 35.919: Document 5: Design and evaluation report*. URL: www.3gpp.org/DynaReport/35919.htm.

[6] 3GPP. *Technical Specification TS 33.401: Security Architecture*. URL: www.3gpp.org/DynaReport/33401.htm.

[7] 3GPP. *Technical Specification TS 35.215: Document 1: UEA2 and UIA2 specifications*. URL: www.3gpp.org/DynaReport/35215.htm.

[8] 3GPP. *Technical Specification TS 35.216: Document 2: SNOW 3G specification*. URL: www.3gpp.org/DynaReport/35216.htm.

[9] 3GPP. *Technical Specification TS 35.217: Document 3: Implementors' test data*. URL: www.3gpp.org/DynaReport/35217.htm.

[10] 3GPP. *Technical Specification TS 35.218: Document 4: Design conformance test data*. URL: www.3gpp.org/DynaReport/35218.htm.

[11] 3GPP. *Technical Specification TS 35.221: Document 1: EEA3 and EIA3 specifications*. URL: www.3gpp.org/DynaReport/35221.htm.

[12] 3GPP. *Technical Specification TS 35.222: Document 2: ZUC specification*. URL: www.3gpp.org/DynaReport/35222.htm.

[13] 3GPP. *Technical Specification TS 35.223: Document 3: Implementors' test data*. URL: www.3gpp.org/DynaReport/35223.htm.

[14] ARM. *Documentation set for AMBA protocol specifications and design tools*. URL: http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ihi0011a/index.html.

[15] A. N. Bikos and N. Sklavos. "Architecture Design of an Area Efficient High Speed Crypto Processor for 4G LTE". In: *IEEE Transactions on Dependable and Secure Computing* PP.99 (2016), pp. 1–1. ISSN: 1545-5971. DOI: 10.1109/TDSC.2016.2620437.

[16] D. H. Bui, D. Puschini, S. Bacles-Min, E. Beigné, and X. T. 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)*. June 2016, pp. 1–4. DOI: 10.1109/ICICDT.2016.7542076.

[17] Ericsson. *Ericsson Mobility Report*. Nov. 2016. URL: https://www.ericsson.com/mobility-report.

[18] ETSI/SAGE. *Document 1: 128-EEA3 and 128-EIA3 Specification*. Dec. 2011. URL: www.gsma.com/aboutus/wp-content/uploads/2014/12/EEA3_EIA3_specification_v1_7.pdf.

[19] ETSI/SAGE. *Document 1: UEA2 and UIA2 Specification*. Mar. 2009. URL: www.gsma.com/aboutus/wp-content/uploads/2014/12/uea2uia2d1v21.pdf.

[20] ETSI/SAGE. *Document 2: SNOW 3G Specification*. Sept. 2006. URL: www.gsma.com/aboutus/wp-content/uploads/2014/12/snow3gspec.pdf.

[21] ETSI/SAGE. *Document 2: ZUC Specification*. June 2011. URL: www.gsma.com/aboutus/wp-content/uploads/2014/12/eea3eia3zucv16.pdf.

[22] ETSI/SAGE. *Document 3: Implementor's Test Data*. Jan. 2011. URL: www.gsma.com/aboutus/wp-content/uploads/2014/12/eea3eia3testdatav11.pdf.

[23] ETSI/SAGE. *Document 3: Implementors' Test Data*. Oct. 2012. URL: www.gsma.com/aboutus/wp-content/uploads/2014/12/Doc3-UEA2-UIA2-Spec-Implementors-Test-Data.pdf.

[24] ETSI/SAGE. *Document 4: Design and Evaluation Report*. Sept. 2011. URL: www.gsma.com/aboutus/wp-content/uploads/2014/12/EEA3_EIA3_Design_Evaluation_v2_0.pdf.

[25] ETSI/SAGE. *Document 4: Design Conformance Test Data*. Jan. 2006. URL: www.gsma.com/aboutus/wp-content/uploads/2014/12/conformance.pdf.

[26] ETSI/SAGE. *Document 5: Design and Evaluation Report*. Sept. 2006. URL: www.gsma.com/aboutus/wp-content/uploads/2014/12/uea2designevaluation.pdf.

[27] Michael Fingeroff. *High-Level Synthesis Blue Book*. Ed. by Mentor Graphics.

[28] A. Gielata, P. Russek, and K. Wiatr. "AES hardware implementation in FPGA for algorithm acceleration purpose". In: *2008 International Conference on Signals and Electronic Systems*. Sept. 2008, pp. 137–140. DOI: 10.1109/ICSES.2008.4673377.

[29] Sourav Sen Gupta, Anupam Chattopadhyay, and Ayesha Khalid. "Designing Integrated Accelerator for Stream Ciphers with Structural Similarities". In: (2012).

[30] P. Hamalainen, T. Alho, M. Hannikainen, and T. D. Hamalainen. "Design and Implementation of Low-Area and Low-Power AES Encryption Hardware Core". In: *9th EUROMICRO Conference on Digital System Design (DSD'06)*. 2006, pp. 577–583. DOI: 10.1109/DSD.2006.40.

[31] S. Hessel, D. Szczesny, N. Lohmann, A. Bilgic, and J. Hausner. "Implementation and Benchmarking of Hardware Accelerators for Ciphering in LTE Terminals". In: *GLOBECOM 2009 - 2009 IEEE Global Telecommunications Conference*. Nov. 2009, pp. 1–7. DOI: 10.1109/GLOCOM.2009.5426313.

[32] S. Hessel, D. Szczesny, N. Lohmann, A. Bilgic, and J. Hausner. "Implementation and Benchmarking of Hardware Accelerators for Ciphering in LTE Terminals". In: *GLOBECOM 2009 - 2009 IEEE Global Telecommunications Conference*. Nov. 2009, pp. 1–7. DOI: 10.1109/GLOCOM.2009.5426313.

[33] NIST. *Advance Encryption Standard (AES)*. 2001. URL: http://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.197.pdf.

[34] NIST. *Recommendation for Block Cipher Modes of Operation*. 2001. URL: http://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-38a.pdf.

[35] NIST. *Recommendation for Block Cipher Modes of Operation: The CMAC Mode for Authentication*. 2016. URL: http://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-38b.pdf.

[36] A. Qamar, F. B. Muslim, F. Gregoretti, L. Lavagno, and M. T. Lazarescu. "High-Level Synthesis for Semi-Global Matching: Is the Juice Worth the Squeeze?" In: *IEEE Access* 5 (2017), pp. 8419–8432. ISSN: 2169-3536. DOI: 10.1109/ACCESS.2016.2635378.

[37] R. Ratasuk, N. Mangalvedhe, and A. Ghosh. "Overview of LTE enhancements for cellular IoT". In: *2015 IEEE 26th Annual International Symposium on Personal, Indoor, and Mobile Radio Communications*. Aug. 2015, pp. 2293–2297. DOI: 10.1109/PIMRC.2015.7343680.

[38] R. Ratasuk, N. Mangalvedhe, Y. Zhang, M. Robert, and J. P. Koskinen. "Overview of narrowband IoT in LTE Rel-13". In: *2016 IEEE Conference on Standards for Communications and Networking (CSCN)*. Oct. 2016, pp. 1–7. DOI: `10.1109/CSCN.2016.7785170`.

[39] R. Ratasuk, B. Vejlgaard, N. Mangalvedhe, and A. Ghosh. "NB-IoT system for M2M communication". In: *2016 IEEE Wireless Communications and Networking Conference*. Apr. 2016, pp. 1–5. DOI: `10.1109/WCNC.2016.7564708`.

[40] B. Schneier, J. Kelsey, D. Whiting, D. Wagner, C. Hall, N. Ferguson, and T. Kohno; et al. *HThe Twofish Team's Final Comments on AES Selection*. May 2000. URL: `www.schneier.com/academic/paperfiles/paper-twofish-final.pdf`.

[41] Synopsys. *DesignWare LTE Security Protocol Accelerator*. URL: `https://www.synopsys.com/dw/ipdir.php?ds=security-protocol-accelerator-lte`.

[42] S. Traboulsi, V. Frascolla, N. Pohl, J. Hausner, and A. Bilgic. "A versatile low-power ciphering and integrity protection unit for LTE-advanced mobile devices". In: *10th IEEE International NEWCAS Conference*. June 2012, pp. 317–320. DOI: `10.1109/NEWCAS.2012.6329020`.

[43] S. Traboulsi, M. Sbeiti, F. Bruns, S. Hessel, and A. Bilgic. "An optimized parallel and energy-efficient implementation of SNOW 3G for LTE mobile devices". In: *2010 IEEE 12th International Conference on Communication Technology*. Nov. 2010, pp. 535–538. DOI: `10.1109/ICCT.2010.5688900`.

[44] Enrique Zabala. *AES deeply explained and animated using Flash*. URL: `www.formaestudio.com/rijndaelinspector/archivos/Rijndael_Animation_v4_eng.swf`.

Appendix A

Register Bit Mapping

TABLE A.1: CONTROL/STATUS register

Name	Bit	Access	Description
Start enable	0	WO	Write a 1 to start a process.
IP ready	1	RO	Security IP is idle.
Mode select	2	RW	Select the processing mode: 0 – Cipher mode 1 – Integrity mode
Software reset	3	RW	Enable the reset of the security IP.
Algorithm select	4-5	RW	Select the algorithm: 0 – N/A 1 – SNOW 3G 2 – AES 3 – ZUC
AES cipher ready	8	RO	0 – cipher not idle 1 – cipher ready
SNOW 3G cipher ready	9	RO	0 – cipher not idle 1 – cipher ready
ZUC cipher ready	10	RO	0 – cipher not idle 1 – cipher ready
AES integrity ready	12	RO	0 – integrity not idle 1 – integrity ready
SNOW 3G integrity ready	13	RO	0 – integrity not idle 1 – integrity ready
ZUC integrity ready	14	RO	0 – integrity not idle 1 – integrity ready
AES confidentiality ready	16	RO	0 – confidentiality not idle 1 – confidentiality ready
SNOW 3G confidentiality ready	17	RO	0 – confidentiality not idle 1 – confidentiality ready
ZUC confidentiality ready	18	RO	0 – confidentiality not idle 1 – confidentiality ready

Printed by Tryckeriet i E-huset, Lund 2018

LUND UNIVERSITY

Series of Master's theses
Department of Electrical and Information Technology
LU/LTH-EIT 2018-649
http://www.eit.lth.se