Key Management Unit for RISC-V Secure Processor

By

FABIO CASTAGNO
S242018

MASTER DEGREE THESIS

Department of Electronic Engineering
Embedded systems
POLITECNICO DI TORINO

Supervisors:

- Andrea Calimera
- Anupam Chattopadhyay
- Jawad Haj-Yahya

DECEMBER 2018

Abstract

Nowadays many embedded applications need to be secure and demand low power operations. The power consumption of designs varies with the implementation choices made by designers. Speed increasing is a key parameters for today's processor but usually this means more power consumption. A trade-off between these 2 parameters must be found.

Secure communications is mandatory when we have to deal with cryptographic mechanisms, the secure key management is one of the top aspect and must be taken into account. Even a well elaborate security concept become weak if the key management is not properly handled. One way of providing isolation between the secure environment and the non-secure environment by using the TEE. The Trusted Execution Environment must be protected with its software structure and this is one of the main research area in cyber-security field. A good way to do this is with the use of Physical Unclonable Functions (PUF).

The operating principle of these devices is pretty simple, it exploits the variability of a device and extracts a unique identifications. It is relatively easy to evaluate and behaves like a random function. Thanks to this property this function is unpredictable and so very difficult to reproduce even when there is a direct access to the system and the functionality is known.

This thesis work involves the development of the Key Management Unit (KMU) based on a new PUF module. This component is responsible for the generation and distribution of the cryptographic keys. The starting point for the development of the Key Management Unit is the design of the single components, each of them has been then tested alone before using inside the KMU. This approach enables to exhaustive test a single instance being sure that the desired behavior is the expected one. Next step is the Synthesis and Implementation phase, where an analysis of different implementation choices with different constraints is made. The provided results including energy are compared to evaluate the savings as well as the latency and area. We prototype our design on NEXYS 4 DDR FPGA. Results show up-to 89% dynamic energy reduction in design with a saving in area.

DEDICATION AND ACKNOWLEDGEMENTS

The first person I want to thank is my advisor, professor Andrea Calimera, for the possibility who gave to me to go to Singapore and for the support during the study and the preparation of this thesis.

I would also like to thank my other thesis supervisors in Singapore: Prof. Anupam Chattopadhyay and Jawad Haj-Yahya.

My sincere thanks also goes to my team colleagues for the time spent together during this internship and for helping me with the project.

I thank my old and new friends who have always encouraged me during this experience, for the trip that we took together and for all the fun that we had.

A special thanks also goes to my family. They always supported me during mine university years and also throughout my life.

TABLE OF CONTENTS

List of Figures

CHAPTER 1

INTRODUCTION

IoT security is growing rapidly and users realize more their vulnerabilities. Studies say that by 2020 there will be more than 20.7 billion connected things. Security is now a priority for the developers but, somehow, people are worried about implementing security features. They think that these new functionalities could hurt speed, performance and battery life. The principle of the secure processor is to make cracking and reverse engineering of the firmware running on it a very hard, long and expensive task. The aim of this project is to build ITUS secure processor. The processor is based on RISC-V ISA, and will include several security features, among which:

- **Platform integrity**
  This will be achieved by secure/authenticated boot
- **Secure storage**
  Several features for securing the memory, beside memory encryption, ITUS includes advanced memory protection features such as, memory integrity and oblivious RAM (ORAM)
- **Isolated execution**
  To enable running trusted applications as secure world, ITUS is crafted with Trusted Execution Environment (TEE) that is achieved with dedicated co-processor
- **Secure IO and Debug**
  ITUS enabled secure IO and Debug protection
- **Device identification and authentication**
  All ITUS Devices will have unique identity that will be assigned to them after fabrication, ITUS will enable authentication process for the device

Physical side channel attacks are not considered for ITUS and will be left for future generation.

To better understand the nature of this project a quick overview is presented here.

- **ITUS Secure SoC**
  - Based on the modern RISC-V architecture, a fast growing eco-system with industry support (Samsung, Nvidia, etc.)
  - The Soc maintain low power design by adding power-management features and optimizing IPs for low power
  - ITUS will include state of the art security features that guarantees system confidentiality, integrity and availability
- **Components in proposed ITUS SoC**
  1. Cryptographic Primitives
  2. A Coprocessor for handling the TEE
  3. Secure boot based on PKI
  4. Key Management Unit based on PUF
  5. Memory protection unit (MPU) for handling encryption, integrity and ORAM
  6. Secure debug controller (SDC) and secure IO (SIO) units

The KMU is the core of this project, without this module the cryptographic keys could not exist. In this thesis a new Key Management Unit is proposed. It is based on the use of a new configurable LFSR based PUF. My thesis work is focused on this unit, from the design of the single modules forming the KMU to the Synthesis and Implementation phase. For each step an exhaustive testing phase has been performed to be sure that the desired behavior of each component was the expected one. A thorough analysis of the Key Management Unit and the work related to this module is reported in section 8.

The remainder of this thesis document is organized as follows. Sections from 2 to 7 are dedicated to explain the single components used in ITUS (to have a clearer idea of the whole system). Section 8 describes the KMU and all the components belonging to it are deeply analyzed. Results and analysis are presented are the end of this section. Section 9 gives a quick overview of the SoC used. Section 10 summarizes the results and presents the conclusions.

1.1 System overview

In Figure 1.1 an overall view of the the system is presented. My work is focused on the implementation and testing of the Key Management Unit. The meaning and the characteristics of this unit are presented in section 8.

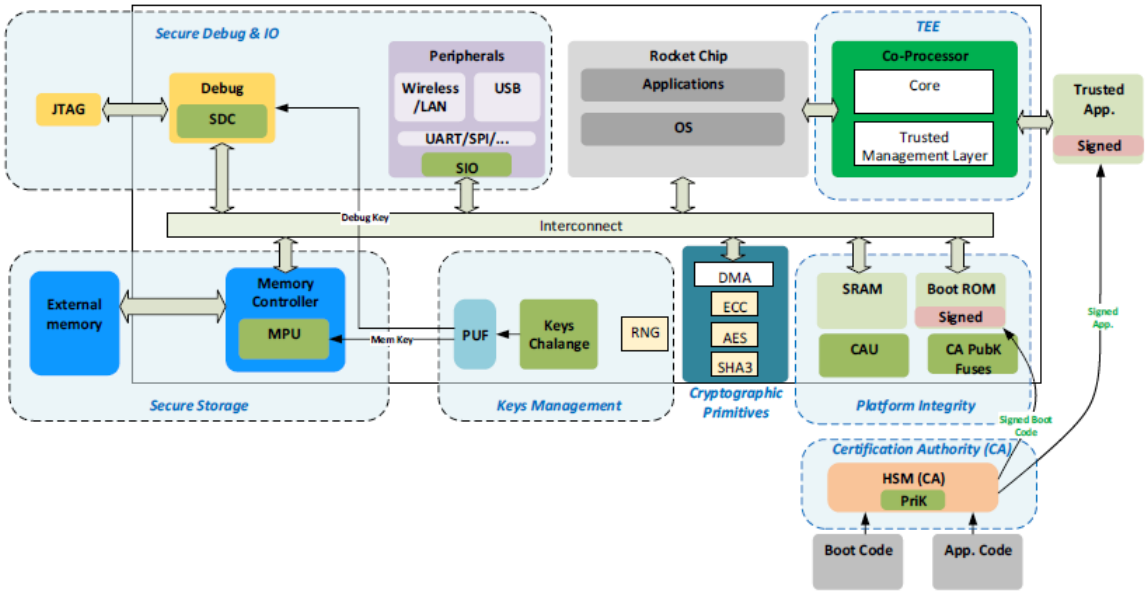

Figure 1.1: ITUS system

The system has the following main features:

1. RISC-V ISA
2. 2 cores, main processor and co-processor (secure processor)
3. Set of IOs
    a. 2 ports of NIC + WIFI
    b. SD-interface
    c. Keyboard
    d. VGA-compatible text display
    e. USB
    f. UART
4. Runs Linux OS
5. Runs JAVA 8

CHAPTER 2

CRYPTOGRAPHIC PRIMITIVES

Cryptographic primitives are the basic blocks used to build security system protocols. Reliability is the fundamental property for this primitives, in fact if they are easily breakable every related protocol will eventually become vulnerable. For example, according to their specifications, if an encryption routine results breakable with a lower number of computer operations than the declared one, that primitive results to fail.
The combination of cryptographic primitives results in the creation of security protocols.
The cryptographic functions employed in the ITUS project are quickly presented in the following subsections.

2.1 Secure Hash Algorithm 3 (KECCAK-256)

KECCAK algorithm was selected by NIST as the winner of the SHA-3 Cryptographic Hash Algorithm Computations in 2012 and SHA-3 was later announced as the hashing standard in 2015. The SHA-3 family is composed of 4 cryptographic hash functions (SHA3-224, SHA3-256, SHA3-384 and SHA3-512). In this project, SHA3-256 is chosen to perform the hashing function in the secure processor.

Two parameters characterize the Keccak function: bitrate r and capacity c. The sponge construction for hash function is as depicted in the following figure, where Pi are the input and Zi are hashed output. The width of the Keccak-f permutation used in the sponge constructions is determined by the sum of r + c. For 256-bit hash output, that is required in this project, the bit rate and capacity of r = 1088 and c = 512 is required.

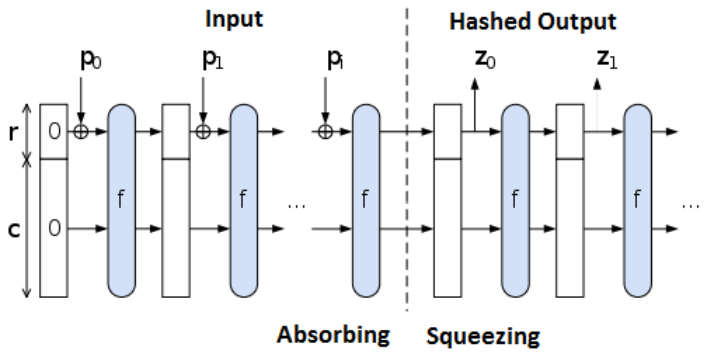

Figure 2.1: Sponge Construction for Hash Function

The sponge construction of Keccak hash function can be divided into three phases: initiation, absorbing and squeezing phases. The first phase initialize the state matrix with zero. In SHA-3 we have 5 x 5 array of 64-bit words (1600 bits). Next, in absorbing phase, there is an XOR of the r-bit with the current matrix state and followed by a series of 24 round of Keccak block transformation (compression functions). Last, the squeezing phase enables the truncation of the state matrix into the final output hash length.

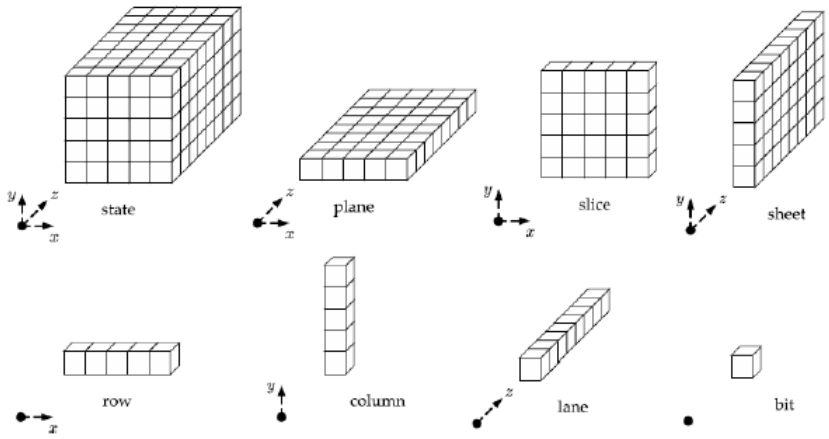

Figure 2.2: State Matrix

Keccak compression function consists of five steps: **theta** (θ), **rho** (ρ), **pi** (ϕ), **chi** (χ) and **iota** (ι). These computations are permutation that uses Boolean logics XOR, AND and NOT. Each of the compression step of Keccak consists of 24 rounds.

- **theta** (θ) The parity of two nearby columns is added (XORed) to each column
- **rho** (ρ) All the lanes are bitwise rotated by a defined offset
- **pi** (ϕ) The 25 lanes are transposed in a fixed pattern, where the bits of each slice are permuted
- **chi** (χ) The 5 bits of each row are non-linearly combined using AND fates and inverters and the result is added to the row
- **iota** (ι) A round constant is added (XORed) to a single lane

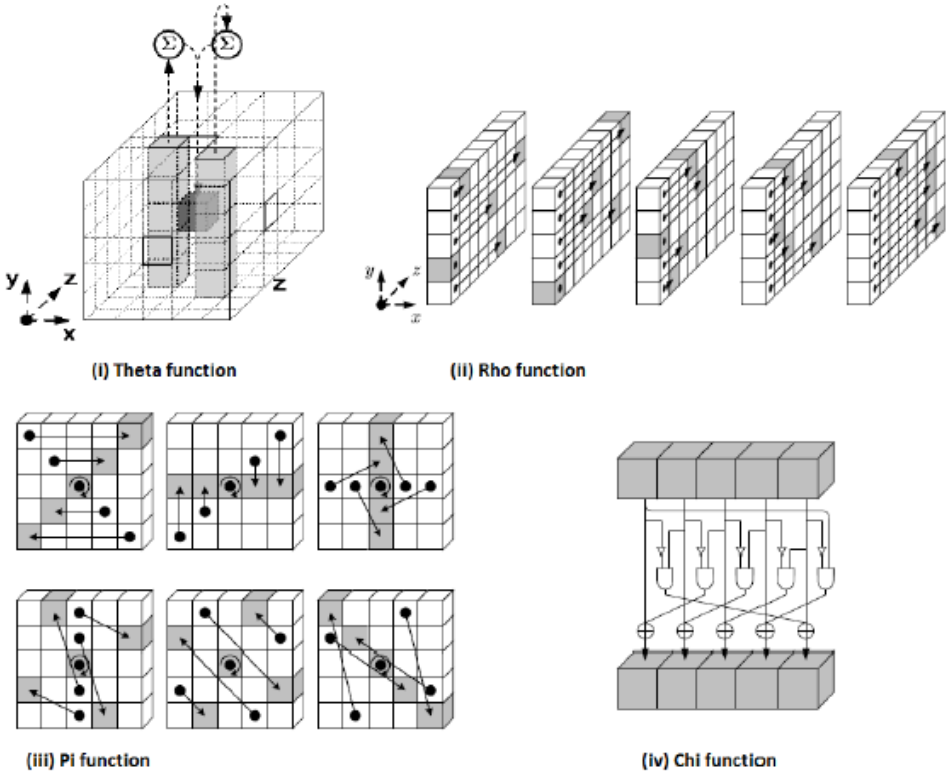

Figure 2.3: Keccak Compression Functions

In terms of hardware implementations, the proposed datapath of Keccak is a depicted as following figure. The Matrix A represent the state matrix on which the Keccak block transformation will take place. For efficient architecture design, Keccak compression steps are combined during the implementation as well. **theta** (θ), **rho** (ρ), **pi** (ϕ), **chi** (χ) and **iota** (ι). There are three Boolean equations in **theta** (θ) step, of which all can be implemented using 5x64 LUT primitives. The LUT is instantiated for 64 times to compute five XOR of 64-bit operands of the equation.

Next, the last two equations of theta step can be combined and executed using 25x64 instantiation of LUT primitive. The step is ended with one bit rotation which can be performed through rewiring. **rho** (ρ) and **pi** (ϕ) permutations can be obtained by simply hardware rewiring. In this case, no hardware resources is required. The constant r [x,y] is fixed and known for each position of Matrix A, also in this case is implemented by means of rewiring.

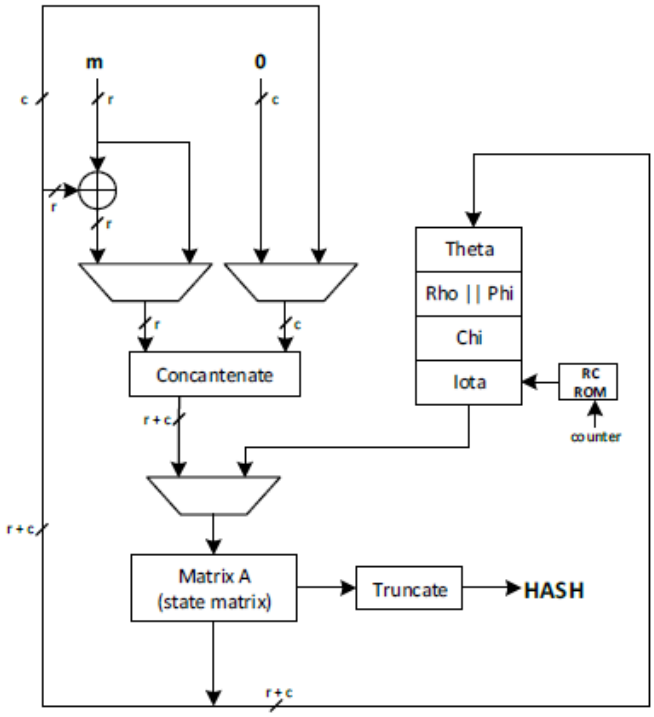

Figure 2.4: SHA-3 (Keccak) Hardware Architecture

chi (χ) step uses 3 logical operations: XOR, NOT and AND. The final step **iota** (ι) is an XOR of the least significant 64 bits of Matrix A with a round constant. The round constant (RC) are stored in a ROM memory.
This Keccak round is composed of 5 steps and it is performed in one clock cycle.
After the completion of a message block (24 rounds), the resulting r-bits of state of Matrix A and the next message block are XORed together. This process is repeated until the last message block has been completed. Last but not the least, at the final state the Matrix A is truncated accoriding to the desired length of the hash output.

2.2 Advanced Encryption Standard (AES-128)

The NIST established the Advanced Encryption Standard (AES), also known as Rijndael, as a specification for encryption of electronic data in 2001.
The processed data blocks length is 128 bits while key length can be different (128, 192 or 256 bits). These lenghts lead to a 3 different AES variants respectively named AES-128, AES-192 and AES-256. In this project, we use the AES-128, this means a key lengths of 128-bits and a number of computation rounds equal to 10.
The first step is the copy of the input into an array (State). State is a 16 bytes array made of 4 rows and 4 columns. The encryption begins by copying the input to an array, named as State. The State is an array of 16 bytes arranged in four rows and four columns (4x4). On each of the State blocks a transformation is performed.

The input value in the initial round is then added to the initial key. After this, 9 identical rounds take place (the final one is a little bit different with respect to the others). The AES computation round (9 rounds) is made of four transformations:

- SubBytes
- ShiftRows
- MixColumns
- AddRoundKey

The final round differs from the one above because the MixColumns transformation is missing. The following figure shows the alghoritme structure.

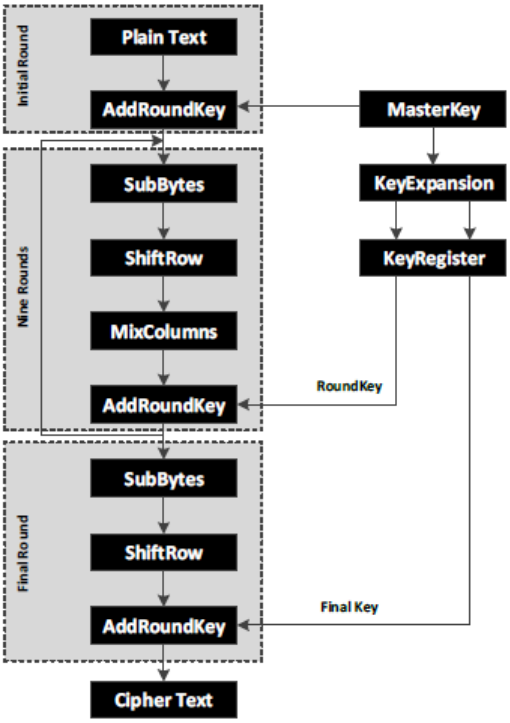

Figure 2.5: AES-128 Encryption Algorithm

2.2.1 SubBytes

A substitution box (S-box) is used by symmetric key cryptography to achieve confusion and diffusion properties. For AES during the first transformation (SubBytes) the S-box is performed. The S-box replaces each $S_{r,c}$ byte in the State matrix with a new $S'_{r,c}$ subbyte.

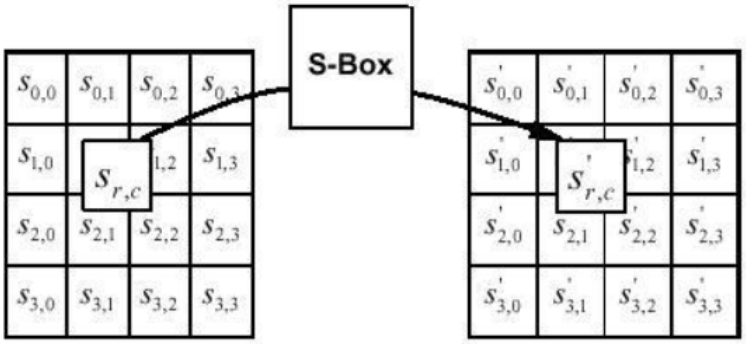

Figure 2.6: SubBytes Transformation

2.2.2 ShiftRows

ShiftRows transformation operates on the State rows, here the bytes are shifted according to an offset. In particular the offset associated to each row is 0, 1 and 2 respectively (first row is not shifted). This step is important so as to avoid the columns in the state being linearly dependent.

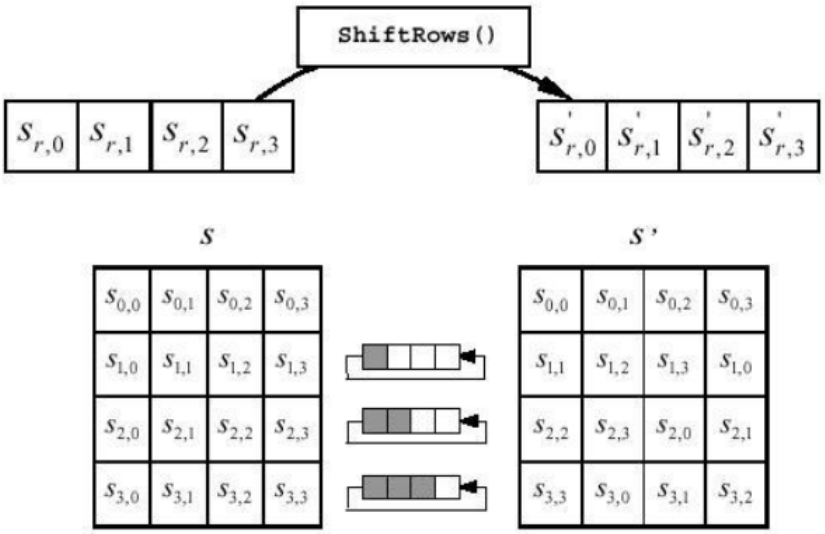

Figure 2.7: ShiftRows Transformation

2.2.3 MixColumns

This invertible linear transformation is the matrix multiplication among each State column. It is composed of multiplication and then addition of the entries.

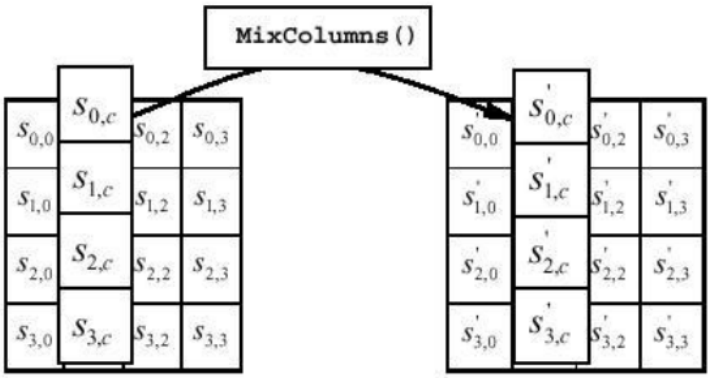

Figure 2.8: MixColumns Transformation

2.2.4 AddRoundKey

In the AddRoundKey the subkey and the state are combined together. Rijndael's key schedule is used to derive the subkey from the main key. An XOR operation is used to add the byte of the subkey with the corresponding byte of the State.

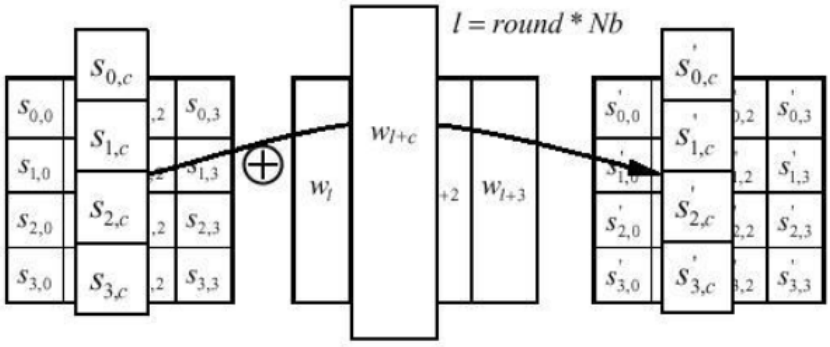

Figure 2.9: AddRoundKey Transformation

The bottleneck of the AES cipher is the SubBytes transformation. To deal with the complex computation problem, our project will look into composite field arithmetic (CFA) technique for AES S-box. Exploitation of CFA enable pure combinational circuit design in order and hence will result in low hardware cost (area and power consumptions).

If a high speed implementation is needed, there is the possibility to imply pipelining. This is a huge advantage in order to speed-up the process. In terms of implementations, the proposed datapath of AES with pipelined CFA S-box is a depicted as following figure.

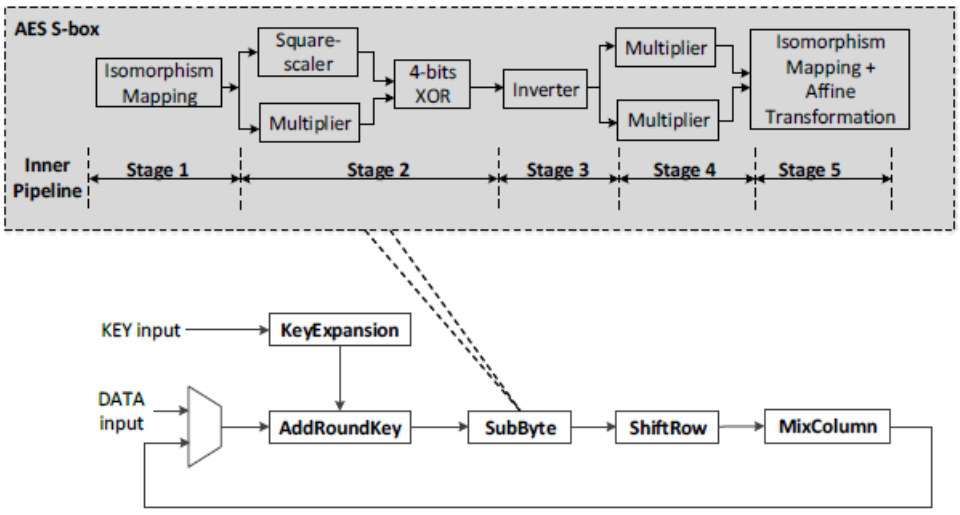

Figure 2.10: AES-128 hardware architecture

2.3 Elliptic Curve Cryptography (ECC)

In addition to symmetric block cipher (also known as private-key cryptographic scheme), public key cryptography also plays important role in providing highly secured channel for digital data transaction. Public key cryptography comes in handy especially in the circumstances where storing numerous symmetric keys is not feasible. Elliptic Curve Cryptography (ECC) is chosen because its security is comparable with respect to the traditional public key cryptographic schemes such as RSA with a much smaller key and is computationally more efficient.
In public key cryptographic scheme, Elliptic Curve Discrete Problem (ECDLP) is defined such that given the key pair (**P, Q**), the scalar point **k** that satisfies $Q = k * P$ is determined. **P** is the base point of the elliptic curve, **k** is the scalar used as the private key and **Q**, the resultant point, is used as the public key. Hence, the ECC security level is determined by the intractability of the ECDLP.
Scalar point multiplication is the most complicated and crucial operation in ECC. This is because it involves a repetitions of point additions and doublings.
A suitable field for ECC computation is the most essential step to taking into account in order to design an ECC hardware module. As an end result, an improved ECC module that is accustomed to suit our processor's the system requirements will be developed in this project.
Two public-key algorithms based on ECC are employed in this project: ECDH (Elliptic curve Diffie-Hellman), employed for key exchange protocol, and ECDSA (Elliptic Curve Digital Signature Algorithm), who plays a role for digital signing.

CHAPTER 3

SECURE BOOT

Secure boot means that at each stage of the boot process a cryptographic check is added. The purpose of this operation is to maintain the boot software image integrity and also prevents an unauthorized or malicious program from running.
A simple implementation would be to encrypt the entire boot code with the private key , known only to Original Equipment Manufacturer (OEM). However, public-key cryptography is slow and the confidentiality of the boot code component is not our interest, we are concerned about the authentication of the executed code. The concept of cryptographic signatures is used here.

3.1 Cryptographic signature protocol

RSA or ECC public-key signature algorithm are the most logical protocol to be applied.
In these protocols a trusted vendor uses a Private key to generate a signature of the code. This signature is pushed in the device with the software binary. The public key of the vendor is contained into the device and can be used to verify the authenticity of the binary code (no modification occurred). The public key must be stored inside the device but there is no need to keep it confidential.
In ITUS we will use the ECDSA for digital signature. ECDSA is a transposition of the DSA (Digital Signature Algorithm) to the elliptic curve. Nowadays it is one of the most widely used numerical signature because is able to offer smaller keys with respect to the same level of security.

Figure 3.1 below illustrated this flow, the protocol details are shown in section **??**.

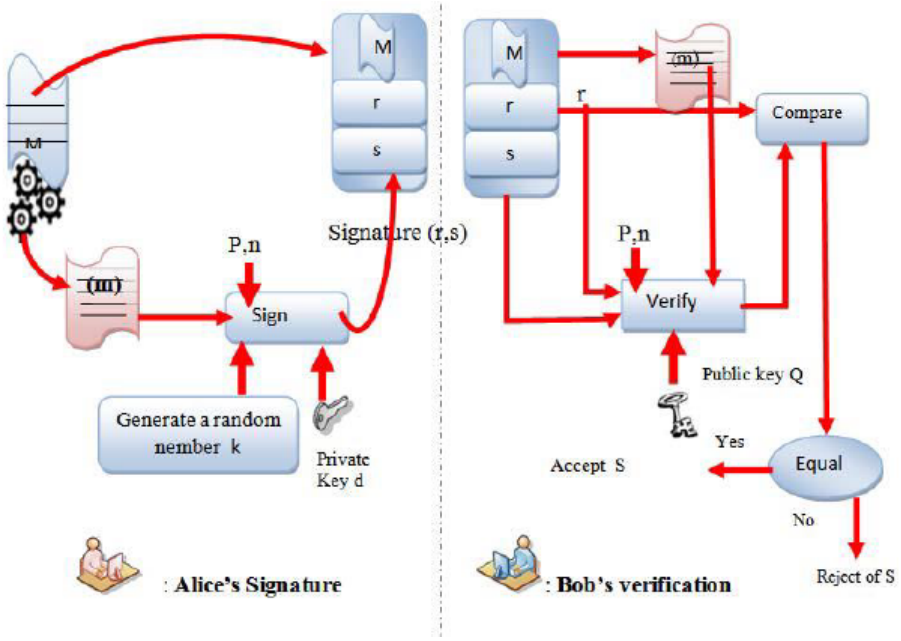

Figure 3.1: Boot code signing and verification flow

3.2 Boot ROM structure

The boot code will be stored in the Boot ROM, the code will be signed as discussed previously, the Boot ROM structure is as followed:

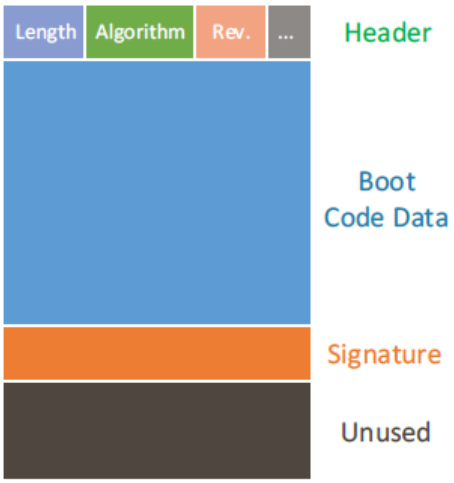

Figure 3.2: Boot ROM structure

At the beginning of the ROM the Header will be located, it will include data about the code size and the Algorithm used for signing and hashing (e.g. SHA2, RSA, etc.), the header will be 4 bytes fixed size. The Boot Code will be located next, the size of the code is as determined in the Length field at the Header. Directly after the boot code, the Signature will be located, its size depends on the algorithm that was used.

3.2.1 Preventing rollback attacks

A rollback attacks is when an attacker tries to update device ROM (by replacing it with other ROM chip in case it's external), for example an older signed boot code with a known vulnerability, we will add a revision number (Rev.) at the Header. Where the at the verification process, we verify the revision number meets the number burned into the chip fuses, or, alternatively the revision number can be stored into TPM's Not volatile memory.

CHAPTER 4

TRUSTED EXECUTION ENVIRONMENT

We can see an execution environment as a software layer running on top of a hardware layer. A device can have two separated execution environment, TEE (Trusted Execution Environment) that will run in the trusted area and a REE (Rich Execution Environment) associated to the untrusted area. The main operating system and the application will be associated to the REE while all the trusted software components will be associated to the TEE.

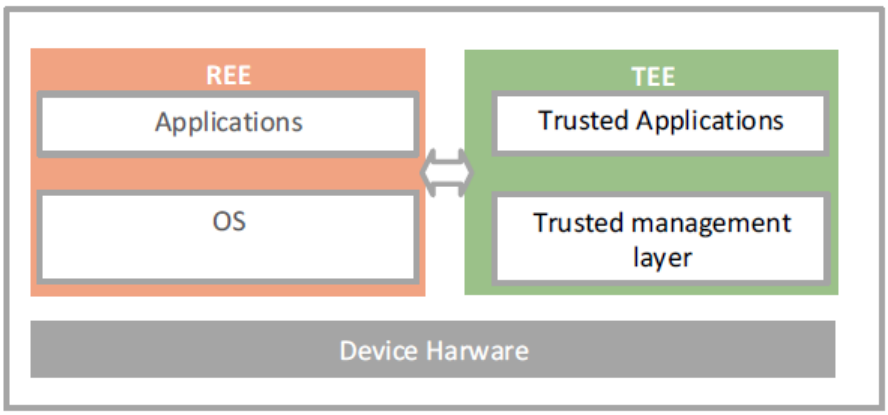

Figure 4.1: Trusted Execution Environment

4.1 TEE components

The TEE is built as co-processor with the main processor (Rocket-chip) as shown in Figure 4.2, the main-processor that is running at the REE will request to move to TEE in order to execute trusted application, once transition to the TEE is performed the trusted application will be authenticated be verifying its digital signature, in addition the application will be copied to the secure storage. Once verified and copied to the TEE, the trusted application will run and once finished it will notify the REE.

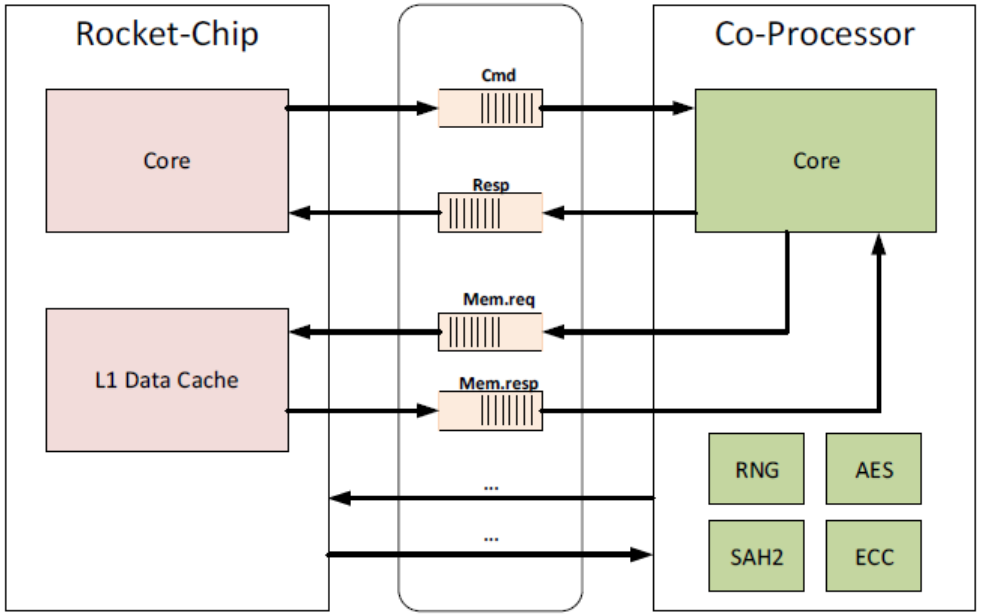

Figure 4.2: TEE as co-processor with the Rocket chip main processor

4.2 The RoCC Interface

The Rocket chip provides dedicated interface, Rocket Custom Coprocessor (RoCC), that can pass instructions from Rocket to the co-processor.
In general, 32-bit RoCC instructions extend the RISC-V ISA and are formatted as shown in Figure 4.3.

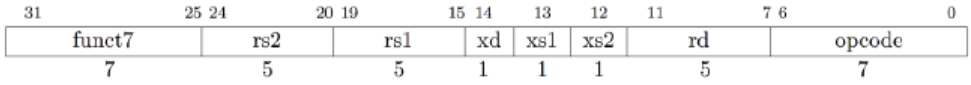

Figure 4.3: The RoCC instruction encoding

The $xs1$, $xs2$, and xd bits control how the base integer registers are read and written by the coprocessor instruction. If $xs1$ is a 1, then the 64-bit value in the integer register specified by $rs1$ is passed to the coprocessor. If the $xs1$ bit is clear, no value is passed over the RoCC interface. The $xs2$ bit similarly controls whether a second integer register specified by $rs2$ is read and passed to the RoCC interface. If the xd bit is a 1 and rd is $not\ x0$, the core will wait for a value to be returned by the coprocessor over the RoCC interface after issuing the instruction to the coprocessor. rd specifies the integer register where the value will be written.
If the xd is 0 or rd is $x0$, the core will not wait for a value from the coprocessor.

CHAPTER 5

MEMORY PROTECTION

In order to run a trusted software an trusted environment, a cryptographic protection of memory is mandatory and essential.
An assumption is made, only the DRAM untrusted is included in the security perimeter of the processor package.

5.1 Memory Protection Unit

Memory Protection Unit (MPU) is a hardware unit, whose role is to protect freshness, integrity and confidentiality of the processor and DRAM traffic over some memory range.
MPU delivers the required protection for the DRAM. Its design is based on the following components:

- Cryptographic primitives that realize the encryption
- An integrity tree
- Message Authentication Code (MAC)
- Anti-replay mechanism

In addition, the MPU support Oblivious RAM (ORAM) for higher memory protection.

CHAPTER 6

SECURE DEBUG

This secure processor provides a method to control the JTAG access, the debug access control is done by the secure-debug-controller (SDC), we have three access modes as followed:

1. No Debug, No Boundary-Scan (BS)
2. No-Debug, allow BS – No debug allowed, BS allowed
3. Secure-Debug - high security
4. JTAG Enabled. Low security (default)

One Time Programmable fuses enable to configure the JTAG mode.

6.1 Secure-Debug mode

JTAG access is performed by using challenge/response-based authentication mechanism, SDC is in charge to perform the access control. JTAG port access is checked every time. If a device is not authorized to access the JTAG, the SDC denied the access attempt.
External debugger tools are needed for this debug feature. Usually the JTAG mode is not enabled on development boards while it is enabled during design manufacturing.

6.2 Debug flow when Secure JTAG mode is enabled

JTAG access authentication uses the challenge/response mechanism (a secret response key is associated to a challenge value). eFuses inside the chip have the task to store the keys.
This is the authentication process during Secure Debug:

- challenge key shifted through the test data output by the JTAG (target side)
- response key generated by the debug tool (host side)
- shift back of the response key through test data input
- if there is a match between the shifted key and the expected internal response, JTAG access is enabled

The flow is described in figure 6.1.

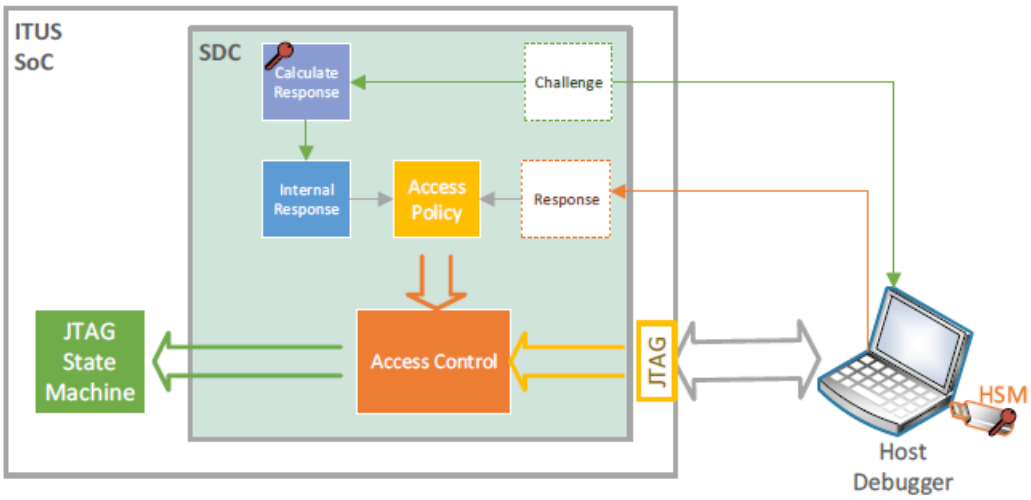

Figure 6.1: Secure Debug mode authentication process

CHAPTER 7

KEYS GENERATION

Cryptographic protocols and algorithms require secret/private keys and, therewith, it is needed to generate, store and manage the necessary keys on embedded devices. Typically, keys are generated based on physical or seeded pseudo-random number generators and are then permanently stored in a protected memory area (e.g. nonvolatile memory (NVM)).
Embedded systems often do not provide a secure memory. Separate security ICs that would offer secure key management and memory are typically not available, commonly due to cost limitation requirements especially in Internet-of-things devices (IoT).

7.1 Public-Key Infrastructure (PKI) and Secure Boot

In ITUS project a PKI is employed to establish system authenticity. Two purposes are achieved:

a. Trust for execution of previously boot modules, this operation is performed during the boot phase

b. Authenticate service request like change or update the firmware or the boot modification

PKI is made of:

- digital certificates (certificate authority CA)
- registration authority, whose job is to verify the user identity requesting a certificate
- central directory, for keys index and storing
- certificate management system

7.1.1 ITUS Certification Authority

For ITUS we use a Hardware Security Module HSM as Certification Authority (CAs). The HSM will generate, store and manage the PKI for our processor. We will have public-key for each device, the mapping between the device and the public key will be stored inside the HSM, and each device will have dedicated identity (ITUS-SOC-ID).

7.1.2 Key generation and storage

The root of trust of the system is formed by key pairs. CA is responsible for the keys generation and for sign the allowed operations (legitimate). For each key there is a correspondent public key embedded into the code and used to check if the signature is correct or not. Instead of fusing the public key into the SoC fuses, a hash of the public key will be fused into the SoC fuses to save space.

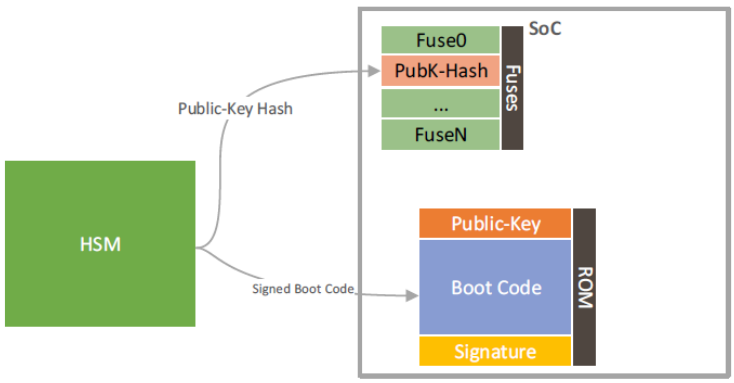

Figure 7.1: Secure Debug mode authentication process

CHAPTER 8

KEY MANAGEMENT UNIT

In ITUS processor we will build a lightweight key generation and management for embedded devices based on Physical Unclonable Functions (PUFs).
This mechanism will be used to generate the following keys:

a. **Symmetric key** – for permanent usage in the embedded system (e.g., for memory encryption)

b. **Asymmetric Private Key** - for permanent usage for device-bound digital signatures (e.g., for public-key based certificates)

c. **Asymmetric session specific private keys** – for short term usage (e.g., in communication protocols, for key negotiation or as ephemeral key in digital signature schemes)

d. **Session specific symmetric keys** – for short term usage (e.g., for data encryption, message authentication in communication protocols)

For the Asymmetric keys the ECC protocol will be used and not RSA.

The main components of the Key Management Unit (KMU) are:

- LFSR based PUF
- True Random Number Generator (TRNG)
- Storage memory
    - Challenges
    - OTP, One Time Programming memory for public key fusing
- Request Handler
- Isolated bus for delivering the keys

The unit structure is reported in Figure 8.1.

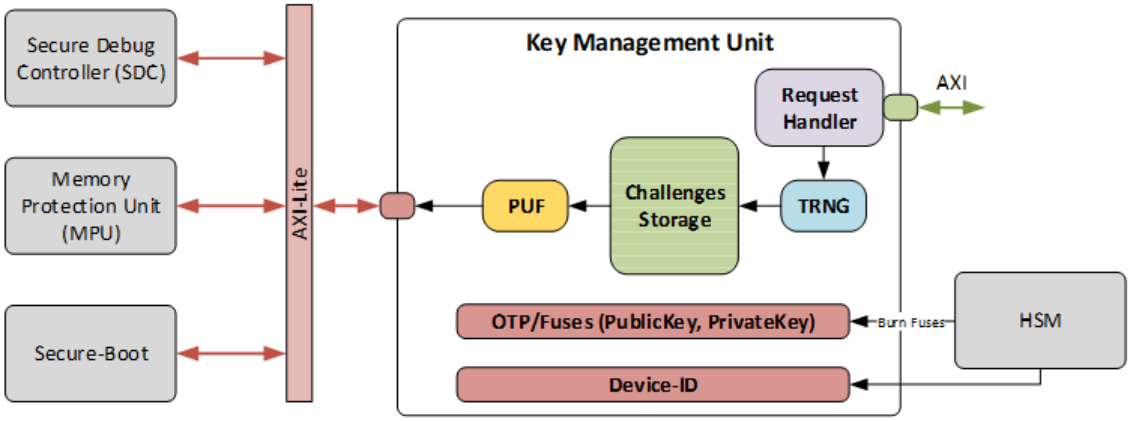

Figure 8.1: Key Management Unit scheme

Each unit will be analyzed and explained in the following sections.
The core of this unit is the PUF module. It accepts 16 bit input data and gives 128 bit data output response. This is the private key that will be distributed through the AXI-lite interface.
The main challenge was to put all this components together and make them work with the Rocket Chip in the proper way.

8.1 Key management for Secure-Debug

Access to a secure-debug is protected by knowledge and proof-of-possession of some kind of credential (numerical pin or password). An embedded system often operates in an unattended way without a possibility to enter a pin or password, so that the credential needs to be stored in a secure way on the insecure embedded system.

8.1.1 Device initialization

A fixed pairing between the target chip (ITUS) and the debug controller is needed and, hence, the establishment of a secure channel connecting both.

This fixed and secure pairing can be achieved with by using the PUF for key provision at the beginning of the device life-cycle and could be realized in the following way:

- The ITUS device is inserted into a secure environment
- The PUF in ITUS is activated, a random challenge C_{dbg} will be entered to the PUF, the output will be K_{dbg} which is the resulting symmetric key, this key is transferred the Host machine and will be permanently stored in the security NVM of HSM at the host
- The ITUS device is removed from the secure environment and ready for operational use
- The challenge C_{dbg} will be saved in non-secure memory inside ITUS

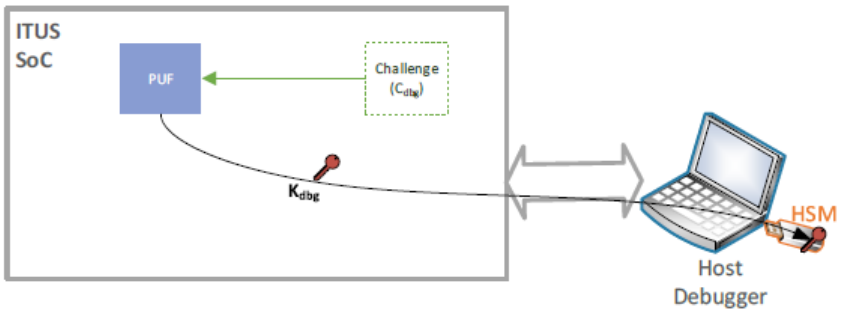

Figure 8.2: Debug key generation process

8.1.2 Debug mode usage

In operational service of the embedded device, PUF is activated on C_{dbg} challenge and produces again the key K_{dbg}.
Using a challenge-response protocol based on a symmetric authentication algorithm AUTH, the ITUS verifies whether the host debugger knows the key.

1. ITUS generates a random challenge C, computes $R_{ITUS} = AUTH(C, K_{dbg})$ and sends C to the host (debugger)
2. On receipt of C the host computes $R = AUTH(C, K_{P,sym})$ and sends R back to ITUS
3. If the received response R equal to R_{ITUS} then the debug is allowed (unlocked) otherwise the debug is prohibited

8.2 LFSR based PUF

In ITUS project and particularly inside the KMU a new configurable LFSR based PUF is employed. In this architecture the response bits from the ring oscillators are selected by the control signals generated from the LFSR states. Using this design, is possible to use the same hardware even if we want to extend the response length (LFSR state numbers are increased). This is possible because the control signals are generated from the LFSR with the given challenge. Any delay-based PUF can adopt this concept of costant-resource scalability.
This new PUF design has a 128-bit response and was tested on NEXYS 4 FPGA (Artix-7). The obtained results are 49.2% uniqueness, 48.5% uniformity and 47.8% bit-aliasing. BCH error correction scheme has been adopted to recover the noisy bits obtaining a final 99.99% reliability response.

8.2.1 PUF overview

The use of secret keys is a must in device authentication and cryptographic applications. Usually a memory can have the task to store a secret key (non-volatile) or an integrated circuit can generate it. Some problems arise when a key is stored into a memory such as safety. A further alternative could be the key generation through a circuit which is irreproducible.
The particularity of a Physical Unclonable Function (PUF) is that the key output depends on the manufacturing variation of the device (timing, delay). In this way there is no mathematical correlation between response and challenge and it is very hard to reproduce the same function. This is the reason that stands behind the word unclonable. Physical variations in delay and logic are the only responsible for the PUF output. A lot of PUF design can be found, each using a different physical variation.
Some limitations can be found in the existing PUFs. For example in some of the delay-based PUF there is a fixed number of response pairs. This happens because of comparator circuit and is related to its size. In this case an increasing of the length corresponds to an increasing of the hardware. In order to find a solution for this problem, this new design is able to generate a new temporary challenge using a Linear Feedback Shift Register (LFSR) whose output is a new input challenge to the PUF.
The new control signals (coming from the original challenge), are derived using the LFSR. In our case a 16-bit challenge produces a 128-bit response. The biggest advantage of this solution is that without any hardware addition, the response length can be extended. Different control signals can be generated can be generated from different states of the LFSR.

Due to the fact that PUF is based on manufacturing variations, same input challenge could lead to a different response, this means different keys. This means that different operating conditions could affects the voltage levels (ambient temperature etc.) introducing some noise. An error correction scheme must be taken into account in order to take action on this problem. The proposed solution is to use the BCH error correction.

8.2.2 Design

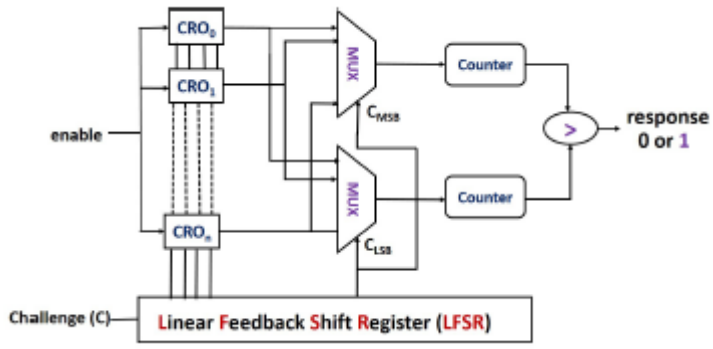

Figure 8.3: Configurable LFSR based PUF

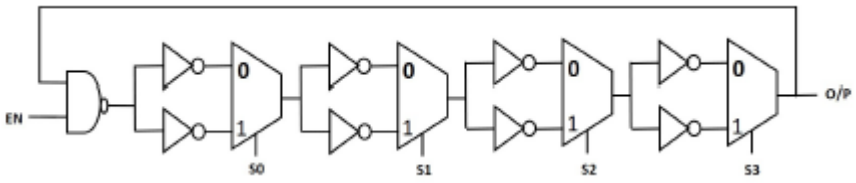

Figure 8.4: Ring Oscillator

In Figure 8.3 we can see the component block diagram. As we can notice the response is generated through the use of ring oscillators.
The Figure 8.4 shows a configurable ring oscillator (5 stages), it is composed of 8 inverters and 4 mpx (2 to 1). The LFSR task is to drive the select signals (S) of the multiplexers. These signals are directly responsible of the operating frequency of the ring oscillator. It possible to notice that different oscillators can achieve different frequencies (as other design), but within the same ring oscillator, different select signals lead to a different frequency. This is the main advantage of this configuration.

This design uses a Linear Feedback Shift Register (the previous state determines the new input bit) with 16 bits parallelism. The linear function of the polynomial is : $x^{16}+x^{14}+x^{13}+x^{11}+1$. A new state is computed every clock cycle. To obtain a consistent response output, the difference frequency between two ring oscillators must be as constant as possible. In order to do this, the LFSR must contain a constant value. During the response generation phase, the LFSR must be in idle state.
256 ring oscillators are needed in this design and they are capable to obtain a 128 bits response. 32 counters are used to derive the response bits.

8.2.3 Experimental results

The Figure 8.5 shows the experimental setup of this module. The FPGA used for testing purposes is the NEXYS-4.

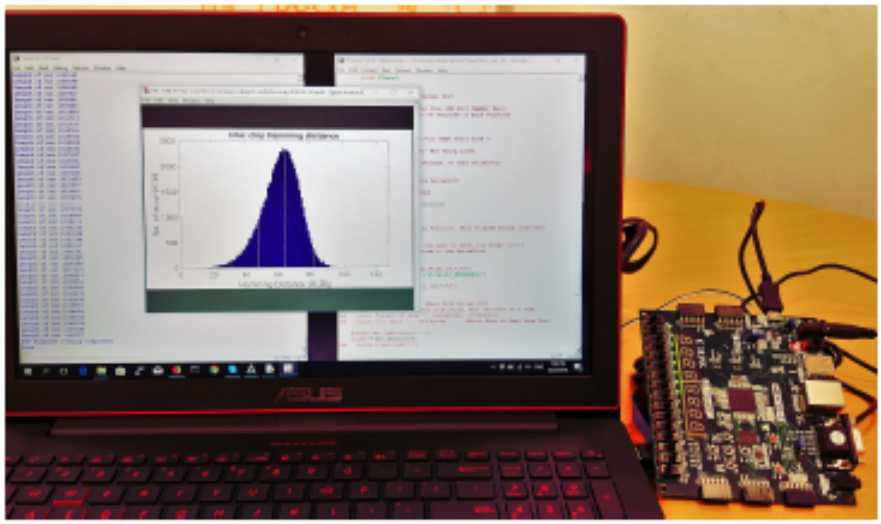

Figure 8.5: FPGA prototype

Each challenge must be associated to a unique response and the device must be able to produce all the possible output combination with a 16 bit challenge. The response distribution can be verified thanks to the Hamming distance plot.

$$IntraHammingDistance = \sum_{i=1}^{k} \frac{HD(R_i, R_{i+1})}{n} * 100 \tag{8.1}$$

k represent the number of challenges while R_i and R_{i+1} are the challenges response.

Figure 8.6 reports the Intra Hamming Distance.

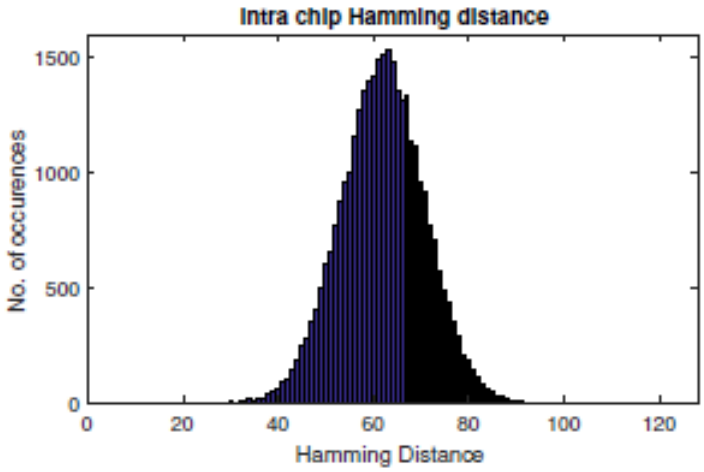

Figure 8.6: Hamming Distance

The response is uniformly distributed (maximum at 64). Noise free and uniformity distribution are some of the parameters that a key must have.
Uniqueness, uniformity, bit-aliasing and reliability are defined as PUF parameters.

A. *Uniqueness*
If we have 2 device with the same configuration, the uniqueness is how much the PUF is able to dinguishthe 2 different outputs. This can be simple read like 2 different device with the same configuration must produce a different output.
Inter chip Hamming distance is used to understand the uniqueness of the design.

$$Uniqueness = \sum_{i=1}^{k-1} \sum_{j=i+1}^{k} \frac{HD(R_i, R_j)}{n} * 100 \tag{8.2}$$

The experimental setup for this purpose involved the use of 2 equal FPGA with the same configuration.

The plot in Figure 8.7 shows the obtained inter chip Hamming distance.

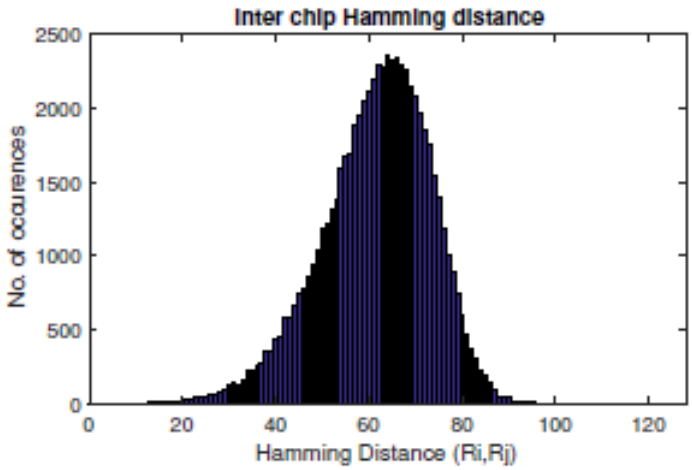

Figure 8.7: Inter Hamming Distance

B. *Uniformity*

The number of 1 and 0 used to create a cryptographic key must be the same. This PUF parameter is known as uniformity. The following Equation 8.3 shows how this parameter is calculated.

$$Uniformity = \frac{1}{n}\sum_{k=1}^{n} R[k] * 100 \tag{8.3}$$

C. *Bit-aliasing*

If, more some reason, a bit is permanently connected to the same logic value (1 or 0), a phenomenon of bit-aliasing occurs. If this situation take place, no matter what the physical hardware is, the produced bit response will always be the same. The Equation 8.4 shows how bit-aliasing is calculated.

$$Bit-aliasing_p = \frac{1}{n}\sum_{i=1}^{k} R_i[p] \tag{8.4}$$

$R_i[p]$ is the pth-bit of the response. 50% of bit-aliasing is the ideal value, this would mean that the output is not correlated to 1 or 0.

D. *Reliability*

Nowadays reliability is considered a must almost in all electronic systems and devices. Reliability means the confidence level on how different situation can reproduce the same response.

$$IHD = \frac{1}{m} \frac{\sum_{t=1}^{m} HD(R_i, R_i^t)}{n} * 100 \tag{8.5}$$

$$reliability = [100 - IHD] * 100 \tag{8.6}$$

PUF reliability is related to the intra Hamming distance. Equation 8.5 and equation 8.6 are used to calculate it. Figure 8.7 shows the PUF reliability (probability density function). Using a 128 bits response, a maximum error of 20 bits arises in this design. An error correction scheme must be used to recover it obtaining th original response.

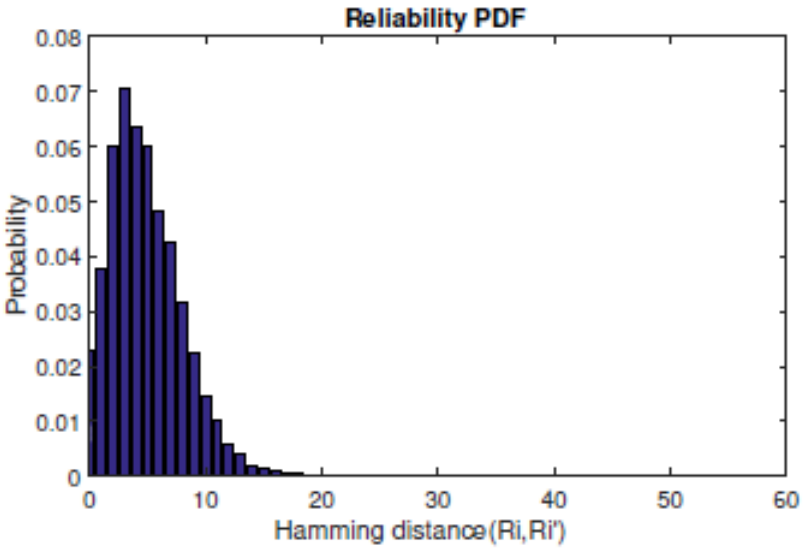

Figure 8.8: Probability Distribution Function

8.2.4 Error correction (BCH)

An error correction scheme is needed in this design because of the temperature variation that can affect the PUF response. Figure 8.9 shows how it works.
This binary error correction scheme uses 3 parameters BCH(n,k,d):

n encoded message

k message length

d error that can be corrected

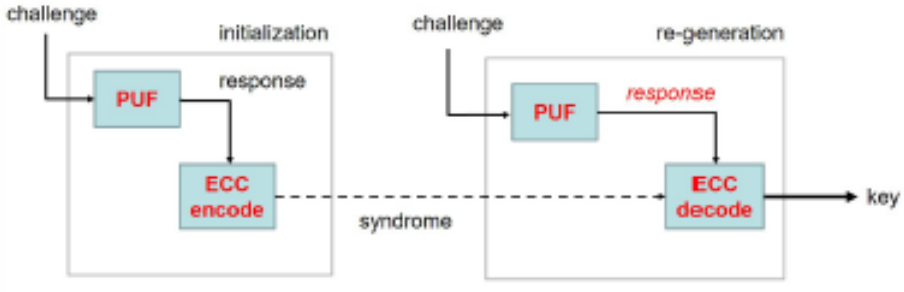

Figure 8.9: PUF error correction scheme

BCH(15,7,2) is employed in this design. This configuration is able to solve up to 2 errors in any position. In particular 5 bits are concatenated to the 128 bits of the key in order to use 19 encoders (7 bit each).
There is the possibility to use 19 pairs of encoder and decoder but in this case a resource sharing approach is adopted. Only one pair is used repetitively. The main advantage of this configuration is the hardware reduction (directly linked to a cost reduction) that reaches the 95%. At the end, the logic is able to correct up to 38 errors (2 errors every 7 bits).
This is the employed polynomial: $(1+x+x^4)(1+x+x^2+x^3+x^4)$.

3 steps are necessary for the BCH decoder:

1 syndrome calculation

2 error locator polynomial (ELP) finding

3 bit correction solving error locator polynomial

If the word associated to the BCH code increase, also the syndromes number increases alongside with the ELP order.

For this design a uniqueness of 49.2%, uniformity of 48.5% and 47.8% bit-aliasing are achieved. This solution can be easily integrated into existing PUF scheme because there is no hardware increasing.
The response is obtained with a 99.99% reliability. This solution can be used both to generate cryptographic keys or for device authentication.

8.3 True Random Number Generator

Application Specific Integrated Circuit (ASIC) were usually the main target to implement the cryptographic algorithms. Nowadays another solution is taking the lead for this kind of applications, FPGA (Field Programmable Gate Array). A lot of reasons are responsible for this kind of choice, in particular an FPGA is reprogrammable, this ensure a lot of flexibility in order to modify and change the algorithm as well as fixing bugs. Moreover the development of an algorithm in ASIC is more difficult and takes more time to be done with respect to a FPGA.
For today's cryptographic systems, a True Random Number Generator (TRNG) is basically mandatory. The typical use of a TRNG are: random sequences generation, keys generation, vectors initialization etc. In a cryptographic system the TRNG is normally responsible of the generation of private or secret parameters. This means that the generation of a random sequence is one of the most important things that must be taken into account and should be unpredictable.
In the KMU a TRNG based on ring oscillators is implied. The output of this component is used as PUF input. The architecture of this component is based on the one published in this paper: *Analysis and Enhancement of Random Number Generator in FPGA Based on Oscillator Rings - 2009*

8.3.1 TRNG Based on Oscillator Rings

The structure of a TRNG is quite simple. It consists of a fixed number of oscillator (they must be equal length) connected to an XOR. The output of the logic gate is the input of a D flip-flop used for sample the incoming signal. The D flip-flop output must be processed to remove bias and increase the entropy from the random signal.
The oscillator rings are responsible for the jitter creation. This jitter is the TRNG entropy source and has the property to have a Gaussian distribution between 0 and 1 (logic low and logic high level) for each clock transition. The jitter is also responsible for the phase drift accumulation in each ring and this lead to a different times transition in the sampling period. The assumption of a uniformly distributed transition region in the sampling period is made.

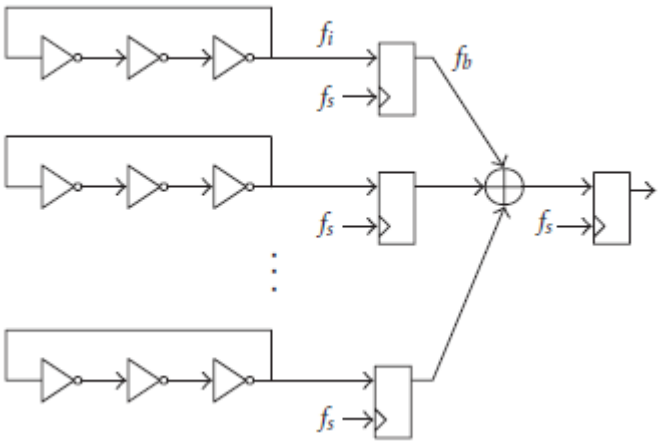

Figure 8.10: TRNG based on oscillator ring

The randomness of the TRNG is improved thanks to the addition of an extra D flip-flop after each ring (Figure 8.10). The overall output randomness is improved thanks to the addition of this D flip-flop without altering the randomness collection of each ring.
The number of the ring inverters must be odds and it directly affects the oscillation frequency (f_i). They are inversely proportional, if the number of inverters decrease, the oscillation frequency increases and viceversa. Small and fast TRNG means that the sampling frequency (f_s) should be as low as possible if compared to the ring frequency.
This configuration ensures that the sampling clock and input XOR signals are synchronous. This is a very important advantage of this configuration. Moreover the setup and hold times in the internal logic of the FPGA is improved.
The switching activity of the XOR-tree inputs is reduced thanks to the adding of the extra D flip-flop. The randomness is collected by the sampling of the free running oscillator rings while the XOR calculation becomes deterministic. Unfortunately a problem of metastability in the D flip-flop can occurs causing its output to be neither in high or low state for a short period of time.

If a transition occurs during the setup and hold time of the D flip-flop, metastability can arises. In order to avoid this situation and the consequently propagation of the metastable state into the XOR tree, another D flip-flop could be added to sample the oscillator ring output. A bigger oscillator rings number lead to a deeper XOR tree logic.

In our case we need a 8-bit output response. This means a 3-level logic depth. This structure cannot be simulated (combinatorial loops) and to reproduce its behaviour an LFSR is implied.

8.3.2 LFSR for simulation

Since a TRNG cannot be simulated as it uses ring oscillators, a Linear Feed-back Shift Register (LFSR) has been implied to check the correct behavior of the other parts of the circuit.
Actually some tool allows the simulation of combinatorial-loops but the correctness would be far from ideal. The only way to test is to make a wrapper top module, implement the design on FPGA and to collect/process data or transfer to a PC for analysis.

A shift register whose input bit is a linear function of its previous state is called LFSR. The most common logic used for the linear function is the XOR (eclusive OR). All the outputs that are connected to the XOR gate are called TAP.
The LFSR must have an initial value different from all zero (XOR with all 0 equal 0 and the input will always be 0) called seed. Whereas the shift register operation is deterministic, also the output values are completely determined (this means that the produced output is pseudo-random). The cycle will repeat after all the register states have been reached. In order to have an output sequence that appears random with a very long cycle, a good feedback function must be chosen.

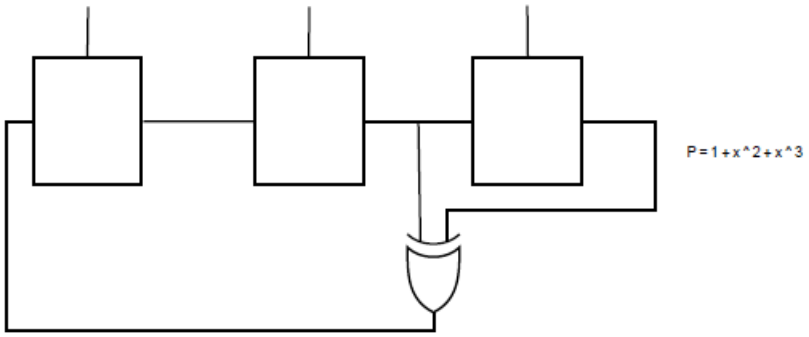

Figure 8.11: Example of an LFSR

The circuit can be initialized with a different seed from Null vector. In the example in Figure 8.11 of the three Flip-Flops are connected to the XOR, which is a input to the other Flip-Flop. Let's suppose that all bits are initialized to '1' after the reset. At each clock cycle the rotation continues and runs a sequence of pseudo random bits on the Flip Flop's outputs, which will be repeated at a given frequency. In this case the sequence will have a length of 7 as shown in the Table 8.1.

Q1	Q2	Q3
1	1	1
0	1	1
0	0	1
1	0	0
0	1	0
1	0	1
1	1	0
1	1	1

Table 8.1: LFSR sequence

It can be demonstrated that the length of sequence is $2^n - 1$. The sequence is often associated to a polynomial where the terms different from zero are those with a position corresponding to the TAP.
In this case $P = 1 + x^2 + x^3$.
LFSRs most typical applications are fast digital counters, pseudo-noise sequences, whitening sequences and pseudo-random number generation.

In Figure 8.12 it is possible to see the behavior of this pseudo-random generator numbers. A clock gate is present for power saving purposes.

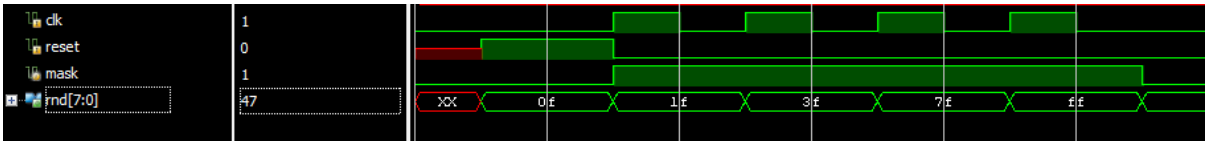

Figure 8.12: Pseudo Random Number Generator

8.4 Challenges Storage

The challenges storage is a memory whose job is to store the challenges coming from the TRNG and the Syndromes coming from the PUF.
At each entry corresponds a different unit that has requested a private key. It is organized in the following way:

- **ID** Unit identifier
- **Challenge** 16 bit coming from the TRNG and used as PUF input
- **Syndrome** 152 bit coming from the PUF and used for the error correction

The PUF Key output is never stored anywhere. Once it is available is only sent to the buffer for the right amount of clock cycles that are needed. In this way the possibilities to stole this information are minimized.
Every time that a unit needs again the same key, this has been recalculated using the previous challenge and syndrome for error correction.
In the following picture is possible to see the interface of this memory. It works on the negative edges of the clock (only element on the KMU), in this way there are no problems of time setting and time hold.

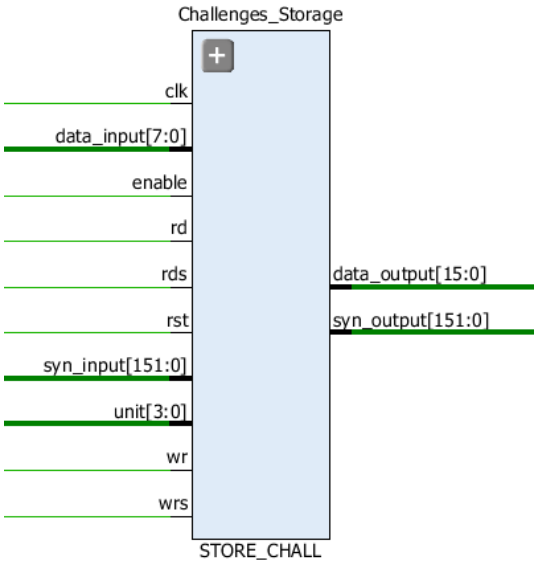

Figure 8.13: Challenges Storage

As you can see the data_input is 8 bit and not 16 bit. This is because the TRNG's output is 8 bit, so the memory needs 2 clock cycle to fill every challenge entry.

8.5 Request Handler

The request handler is responsible of the correct behavior of all the components belonging to the KMU. It is basically a Finite State Machine (FSM) where from each state is possible to go to every other state.
These are the 4 states:

- **IDLE** no operation are requested, this is the waiting state
- **GENERATE** a new challenge must be generated
- **FETCH** a new key must be generated and then sent through the buffer, the output syndrome must be stored back into the storage memory
- **DISTRIBUTE** a key must be re-generated (challenge and syndrome must be sent to the PUF module) and then sent through the buffer

Once the machine goes into a state, the input becomes transparent, that means no matter if there is a request, this will be served only once the current job will be done.

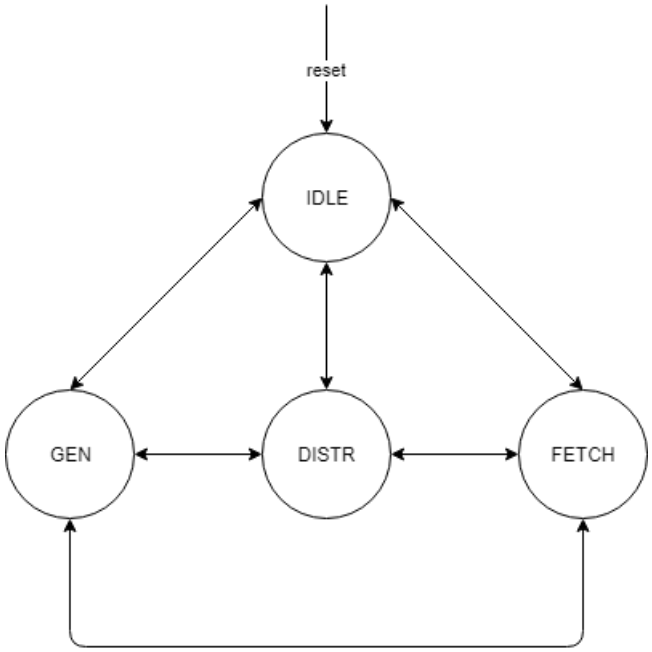

Figure 8.14: Request Handler FSM

8.6 Design approach

The primary objective of the Key Management Unit is to generate the cryptographic keys and distribute them to the consumers (other units) such as Debug, Memory, Boot etc.
In order to accomplish to this task, all the previously seen components are needed. The starting point was the block scheme scheme presented in Figure 8.1. The first thing to do was the design of the single components, each of them has been tested alone before using inside the KMU. This approach enables to exhaustive test a single instance being sure that the desired behavior is the expected one.
The hardware description language used is Verilog while the emplyed software for this project is Vivado, developed by Xilinx.
The main characteristics of the single components are reported here below:

- **TRNG** - based on ring oscillators. It was not possible to simulate its behavior (for this purpose a pseudo-random number generator has been used) so it has been synthesized and physically tested on the NEXYS 4 DDR FPGA. It is composed of 108 ring oscillators (16 for each output bit generated) with an output parallelism of 8 bits. The challenge must be 16 bits so two iterations are needed in order to generate the final output, obtained by concatenating the output of the 2 iterations.

- **Challenges Storage** - this memory has 16 entries (this parameter can be easily modified if more memory lines are needed) and each of them is associated to a single consumer. Every consumer is identified by an ID (4 bits) that is associated to a challenge (16 bits) and the corresponding syndrome (152 bits).

- **PUF** - This module contains both the Col-PUF and the BCH error correction mechanism. The 16 bits challenge is the input of this unit alongside the syndrome (only if needed). The 2 outputs are the 128 bits cryptographic key and the 152 bits syndrome (sent back to the challenge storage)

- **Request Handler** - This FSM (Finite State Machine) controls and manages the previous components and it is responsible for the correct behavior of the whole KMU.

8.7 Simulation results

In order to simulate the overall behavior of the KMU I have used Vivado software. As stated above, it is not possible to simulate the TRNG and the PUF (key generation) so some changed were necessary. The TRNG has been replaced with the LFSR while the key generation module inside the PUF has been replaced with a small memory. The syndrome generation module can be simulated so no changes were applied.

The first step was to test each unit independently and then go up in the hierarchy. In Figure 8.15 we can see a test-bench case where the challenges storage is filled with pseudo random number generated by the LFSR.

It is also possible to notice that for each entry 2 clock cycle are needed to fill the 16 bit.

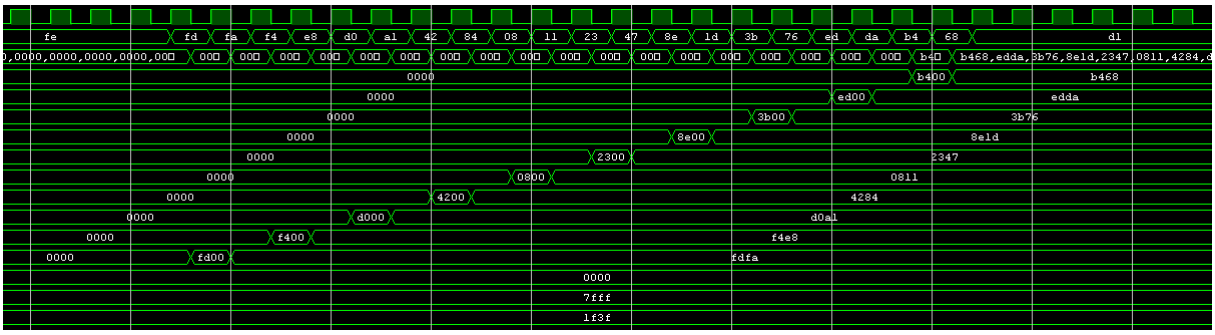

Figure 8.15: Challenges storage filling

As we can see, the challenges storage is sensible to the negative edge of the clock. This choice has been made both for speed improvement and easier management.

In Figure 8.16 the behavior of the whole unit is simulated. As we can see after the key generation request even if another request occurs, it is simply ignored until the previous one has been satisfied.

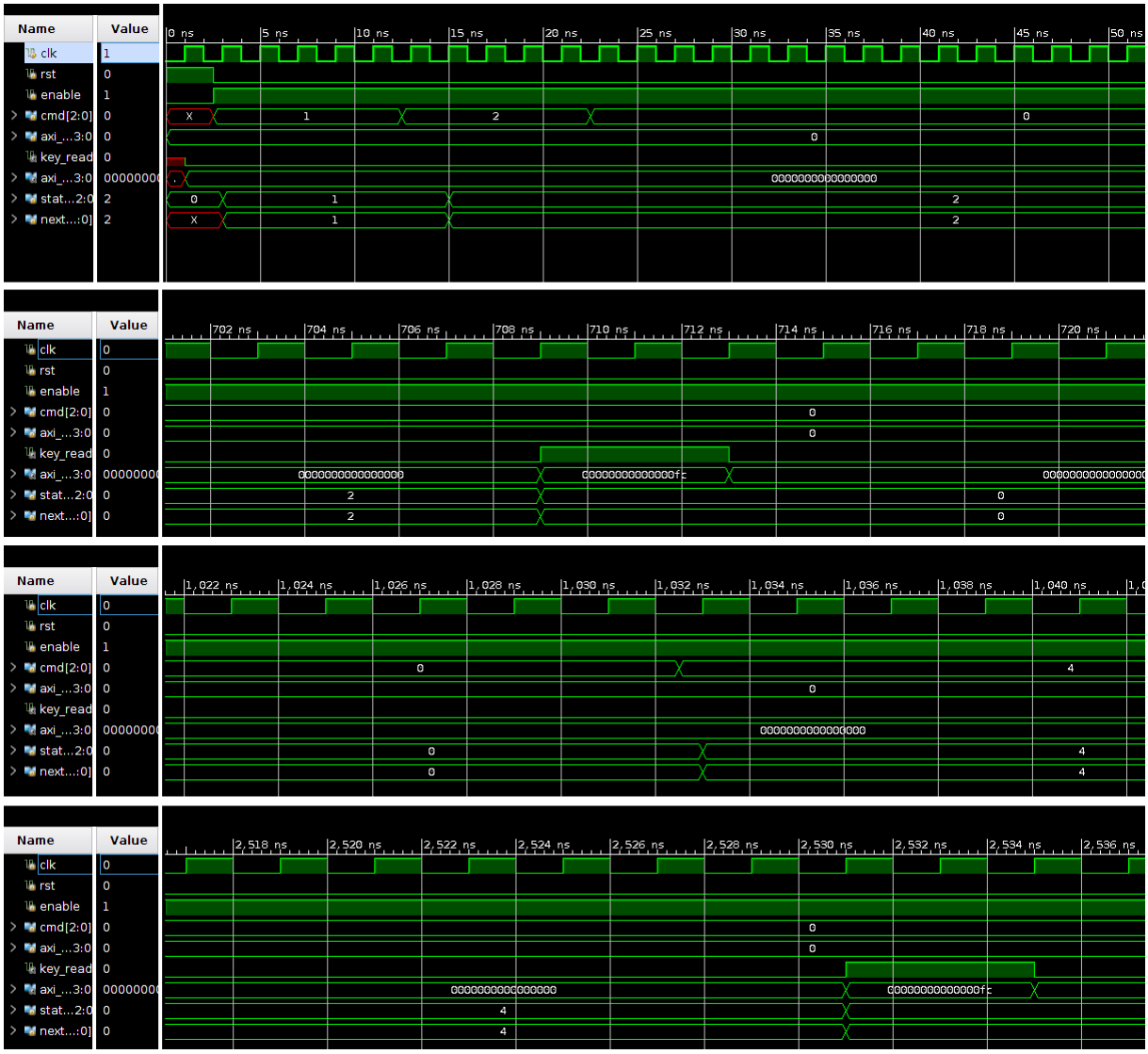

Figure 8.16: KMU behavior

These pictures are captured during the simulation phase. Even if 2 modules cannot be simulated (TRNG and PUF are based on ring oscillators), we can have a realistic idea of the overall behavior of the unit.

The clock constraint of 10 ns is largely met (more accurate measurements are made after the synthesis and implementation phases).

8.8 Synthesis approach

Also in this case the *divide et impera* approach has been followed. For each component a first synthesis took place to have an idea of the consumed power, speed and occupied area. In particular a detailed RTL report is produced in this phase, to see the real FPGA occupation, and to know the characteristics of the single components employed in the unit.
A lot of synthesis have been performed for each component first and for the Key Management Unit. Each time some parameters have been modified to see the behavior under different constraints. In particular the clock constraints was more restrictive from time to time.
The collected data related to power (static and dynamic), slack, number of Slice LUTs and number of flip-flops have been grouped in some tables. The final stage was to take these data and to build some graphs to have a better understand of these quantities behavior.

Once the synthesis was complete, a further action was needed in order to generate the bitstream and consequently physically test the components on the FPGA.
The approach in this implementation phase was very similar to the adopted previous one. Also in this case a lot of different implementation have been performed for each component. Alongside with the clock constraints modification, two more option have been added during this phase: route and placement optimization with power saving.
In this case the design has been modified in order to find the best solution in terms of area and power consumption. The final tables and graphs shows the effects of these choices.

The final step is the bitstream generation and its download on the FPGA. At this point a physical test of the units have been performed with the help of indicator leds or serial UART connection for the output data collection.

8.9 Synthesis results

The synthesis has been done for 3 different modules: TRNG, PUF and the whole KMU. A lot of synthesis have been performed with different clock constraints to see the affections on power consumption and slack. The number of LUTs involved is also a useful parameter to have an idea of the occupied area on FPGA.

During this phase a detailed RTL component usage has been made. All the results are summarized in tables and images below.

8.9.1 PUF synthesis

The results coming from the synthesis phase are grouped in the following table. Starting from a clock constraints of 229 MHz several synthesis have been performed decreasing the clock frequency.

clk[Hz]	Period[ns]	Power[mW]	Dyn[mW]	Stat[mW]	slack[ns]	SLICE LUTs	FF
$2.29 \cdot 10^8$	3	135	51	84	-1.351	670	549
$2.09 \cdot 10^8$	4	122	38	84	-0.351	662	549
$2.00 \cdot 10^8$	5	115	31	84	0.649	596	549
$1.67 \cdot 10^8$	6	109	25	84	1.649	596	549
$1.43 \cdot 10^8$	7	106	22	84	2.649	596	549
$1.25 \cdot 10^8$	8	103	19	84	3.649	596	549
$1.11 \cdot 10^8$	9	101	17	84	4.649	596	549
$1.00 \cdot 10^8$	10	99	15	84	5.649	596	549
$6.67 \cdot 10^7$	15	94	10	84	10.649	596	549
$5.00 \cdot 10^7$	20	92	8	84	15.649	596	549
$3.33 \cdot 10^7$	30	89	5	84	25.649	596	549
$2.50 \cdot 10^7$	40	88	4	84	35.649	596	549
$2.00 \cdot 10^7$	50	87	3	84	45.649	596	549

This results have been plotted to see the behavior of the unit and to better understand how the clock frequency affects the power, slack and LUTs number.

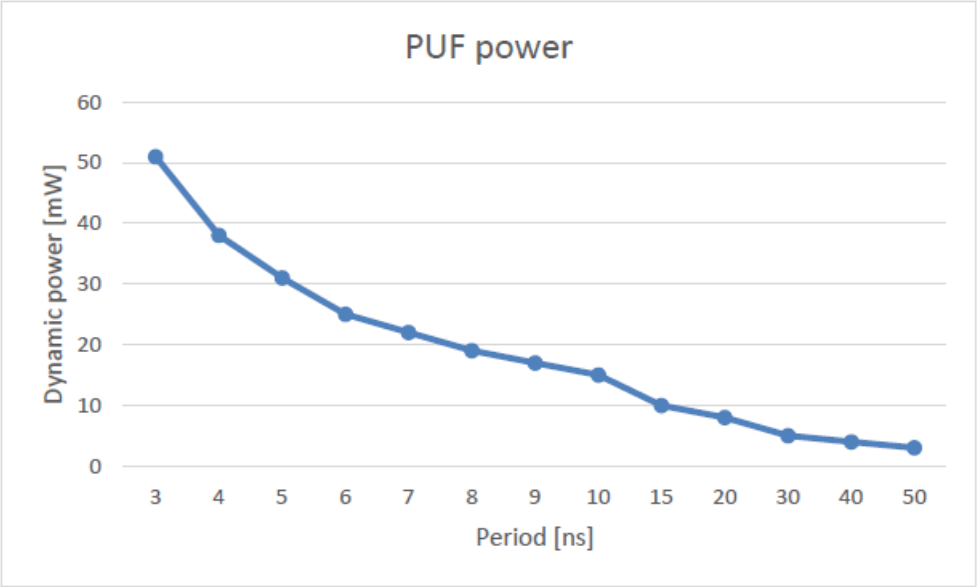

Figure 8.17: PUF dynamic power

This first graph shows how the clock affects the PUF power consumption. In this case only the dynamic power have been plotted cause the static power is still the same (84 mW). As expected stricter constraints led to a higher power consumption and viceversa. Is interesting to notice how fast the dynamic power decreases with respect to the period.

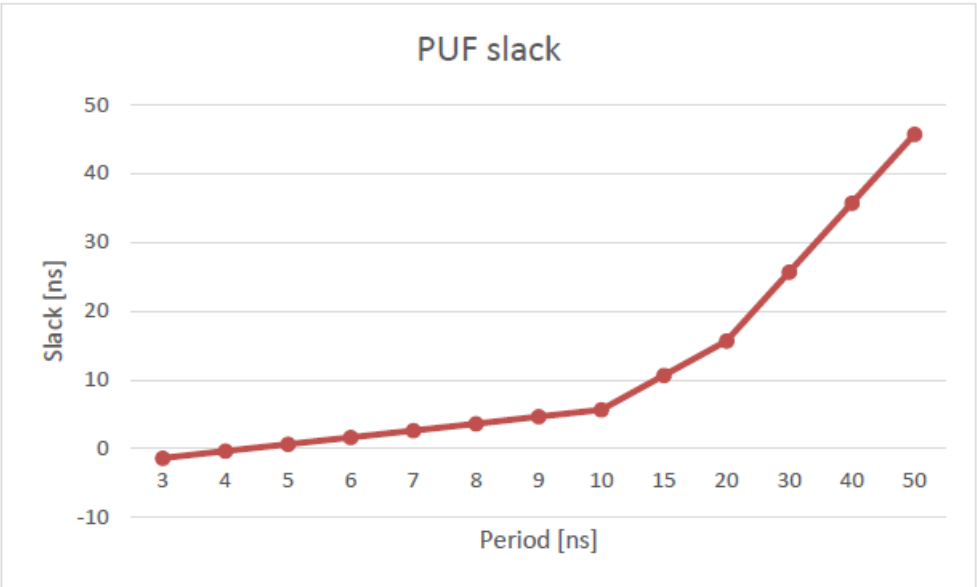

Figure 8.18: PUF slack

The difference between the required time and the arrival time grows linearly. The slack become negative only when the period is shorter than 5 ns. This unit can operate at 200 MHz.

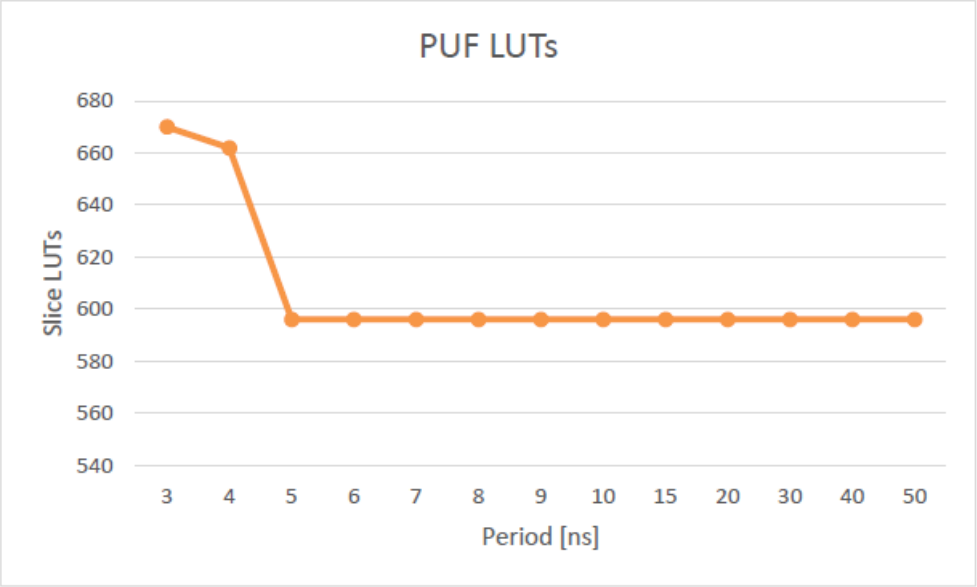

Figure 8.19: PUF LUTs

Once the slack is positive the number of LUTS remains the same for all the configuration (596).

8.9.2 TRNG synthesis

Also in this case the results coming from the synthesis phase are grouped in the following table. In this case in order to get a negative slack I started from a clock constraints of 249 MHz down to 20 MHz.

clk[Hz]	Period[ns]	Power[mW]	Dyn[mW]	Stat[mW]	slack[ns]	SLICE LUTs	FF
$2.29 \cdot 10^8$	2	231	134	97	-0.447	408	136
$2.29 \cdot 10^8$	3	186	89	97	0.553	408	136
$2.09 \cdot 10^8$	4	164	67	97	1.553	408	136
$2.00 \cdot 10^8$	5	151	54	97	2.553	408	136
$1.67 \cdot 10^8$	6	142	45	97	3.553	408	136
$1.43 \cdot 10^8$	7	135	38	97	4.553	408	136
$1.25 \cdot 10^8$	8	131	34	97	5.553	408	136
$1.11 \cdot 10^8$	9	127	30	97	6.553	408	136
$1.00 \cdot 10^8$	10	124	27	97	7.553	408	136
$6.67 \cdot 10^7$	15	115	18	97	12.553	408	136
$5.00 \cdot 10^7$	20	110	13	97	17.553	408	136
$3.33 \cdot 10^7$	30	106	9	97	27.553	408	136
$2.50 \cdot 10^7$	40	104	7	97	37.553	408	136
$2.00 \cdot 10^7$	50	102	5	97	47.553	408	136

Also in this case this results have been plotted to see the behavior of the unit and to better understand how the clock frequency affects the power and slack. As we can notice the LUTs number is not affected by the clock constraint and it is always constant to the value 408.

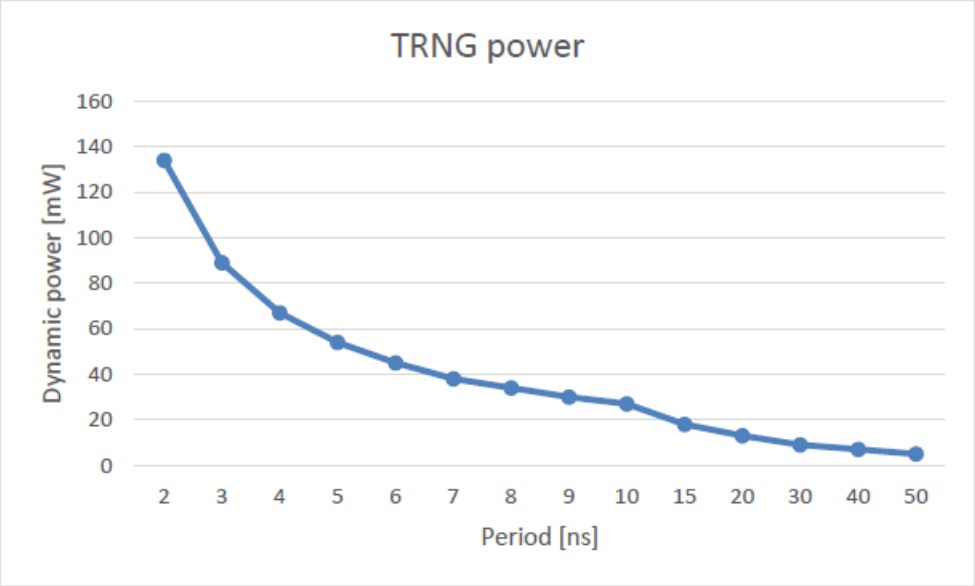

Figure 8.20: TRNG dynamic power

This first graph shows how the clock affects the TRNG power consumption. Only the dynamic power have been plotted cause the static power is still the same (97 mW). As expected stricter constraints led to a higher power consumption and viceversa. Also in this case the dynamic power drops drastically with respect to the period. The power consumption of this module impacts a lot on the performance of the KMU, for this reason a clock gating is present for this module when implemented inside the KMU. In this way a lot of power is saved.

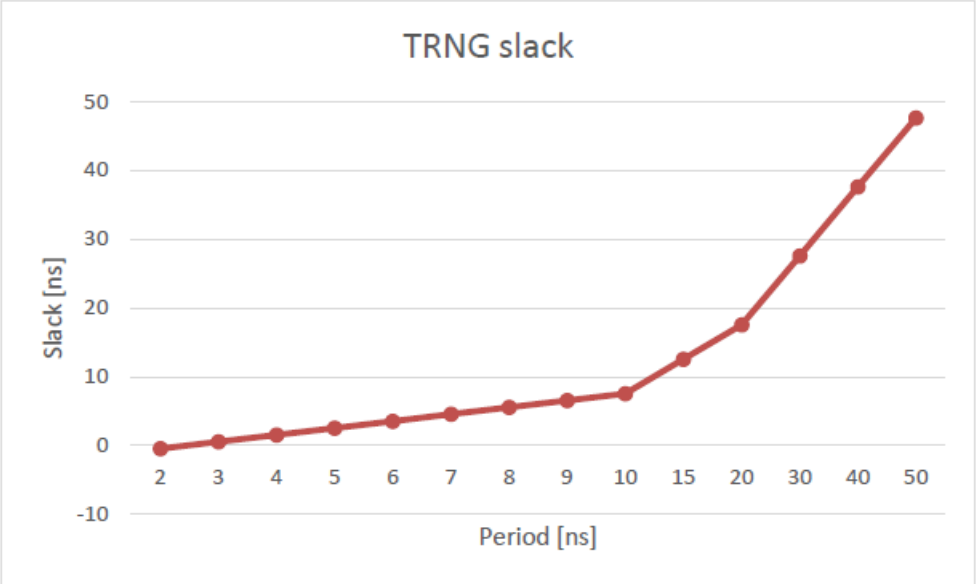

Figure 8.21: TRNG slack

Also here the difference between the required time and the arrival time grows linearly. The slack become negative only when the period is shorter than 2 ns.

8.9.3 KMU synthesis

These are the results coming from the synthesis of the whole KMU.

clk[Hz]	Period[ns]	Power[mW]	Dyn[mW]	Stat[mW]	slack[ns]	SLICE LUTs	FF
$2.00 \cdot 10^8$	5	266	182	84	-0.830	6394	4235
$1.67 \cdot 10^8$	6	236	152	84	-0.330	6394	4235
$1.43 \cdot 10^8$	7	214	130	84	0.170	6322	4235
$1.25 \cdot 10^8$	8	198	114	84	0.670	6316	4235
$1.11 \cdot 10^8$	9	185	101	84	1.170	6316	4235
$1.00 \cdot 10^8$	10	175	91	84	1.670	6316	4235
$6.67 \cdot 10^7$	15	145	61	84	4.170	6316	4235
$5.00 \cdot 10^7$	20	129	45	84	6.670	6316	4235
$3.33 \cdot 10^7$	30	114	30	84	11.670	6316	4235
$2.50 \cdot 10^7$	40	107	23	84	16.670	6316	4235
$2.00 \cdot 10^7$	50	102	18	84	21.670	6316	4235

This results have been plotted to see the behavior of the unit and to better understand how the clock frequency affects the power, slack and LUTs number.

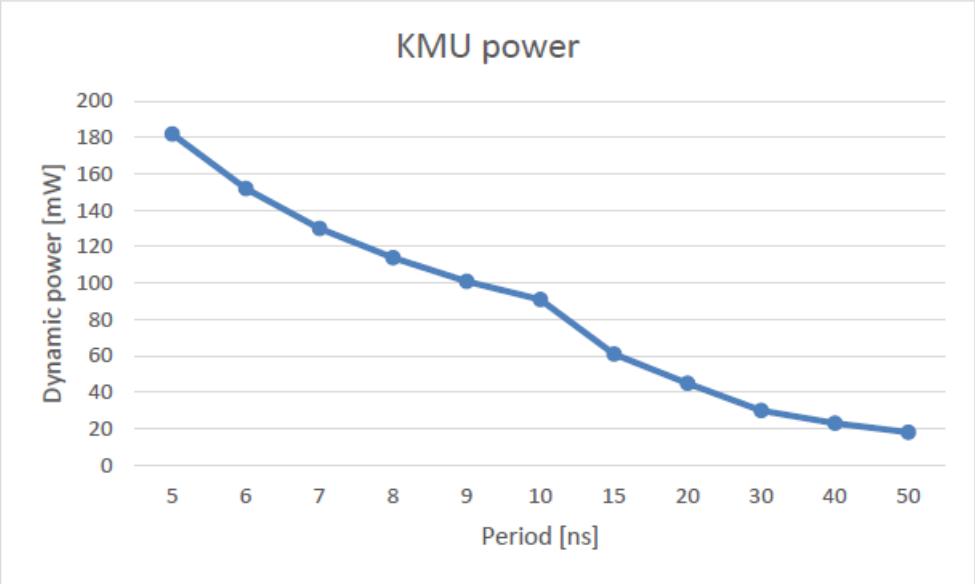

Figure 8.22: KMU dynamic power

This first graph shows how the clock affects the KMU power consumption. Also in this case only the dynamic power have been plotted cause the static power is still the same (84 mW). As expected stricter constraints led to a higher power consumption and viceversa. Also here is interesting to notice how fast the dynamic power decreases with respect to the period. In particular for a period equal to 50 ns, the dynamic power consumption is more than 7 times less with respect to a period of 7 ns.

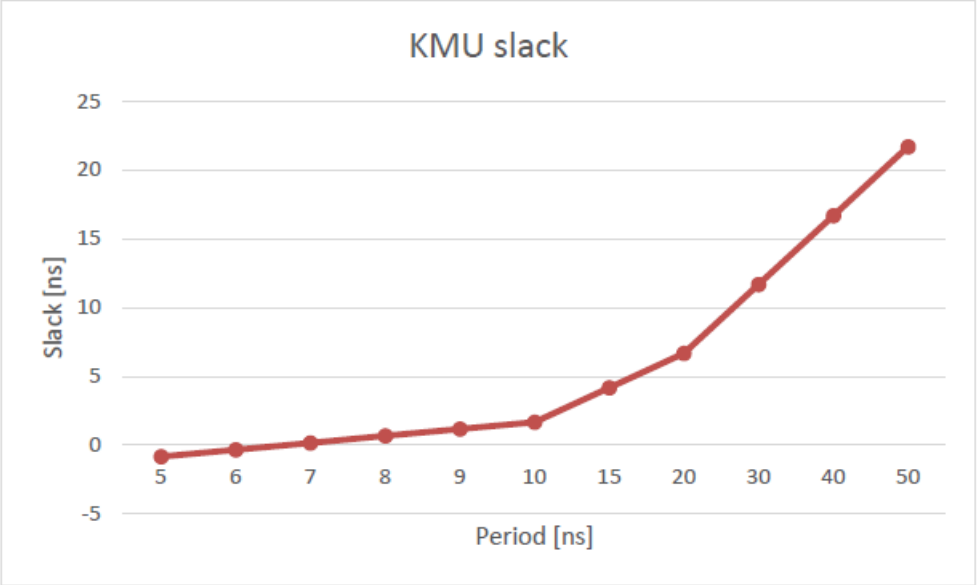

Figure 8.23: KMU slack

The difference between the required time and the arrival time grows linearly. The slack become negative only when the period is shorter than 7 ns (143 MHz).

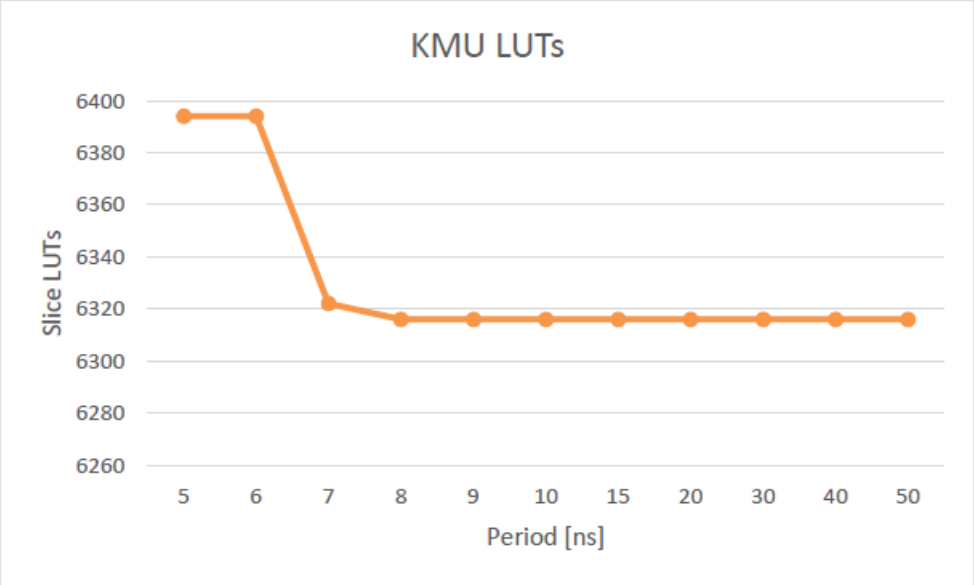

Figure 8.24: KMU LUTs

Also here once the slack is positive the number of LUTS remains almost the same for all the configuration (6316). There is one value that differs from the others when the clock constraint is set at 7 ns of period. In this case 6 more LUTs are needed.

8.9.4 Components usage

A more detailed resource usage of the single units is presented here below. These results came from Vivado software after having performed the synthesis phase.

8.9.4.1 PUF usage

Report

```
Detailed RTL Component Info :
+---Adders :
           2 Input     32 Bit       Adders := 1
           2 Input      8 Bit       Adders := 33
           2 Input      5 Bit       Adders := 7
           2 Input      4 Bit       Adders := 3
           2 Input      3 Bit       Adders := 1
           2 Input      2 Bit       Adders := 1
+---XORs :
           2 Input      4 Bit         XORs := 2
           4 Input      1 Bit         XORs := 1
           2 Input      1 Bit         XORs := 22
```

```
           3 Input      1 Bit         XORs := 6
+---Registers :
                     152 Bit    Registers := 3
                     133 Bit    Registers := 3
                     128 Bit    Registers := 3
                      15 Bit    Registers := 3
                       8 Bit    Registers := 33
                       7 Bit    Registers := 2
                       5 Bit    Registers := 6
                       4 Bit    Registers := 9
                       3 Bit    Registers := 1
                       2 Bit    Registers := 2
                       1 Bit    Registers := 86
+---Muxes :
           2 Input    152 Bit        Muxes := 6
           2 Input    133 Bit        Muxes := 7
           2 Input    128 Bit        Muxes := 7
           2 Input     15 Bit        Muxes := 10
           4 Input     15 Bit        Muxes := 1
           3 Input     15 Bit        Muxes := 1
           2 Input      8 Bit        Muxes := 34
           2 Input      7 Bit        Muxes := 2
           2 Input      5 Bit        Muxes := 27
           4 Input      5 Bit        Muxes := 1
           3 Input      5 Bit        Muxes := 1
           2 Input      4 Bit        Muxes := 9
           4 Input      4 Bit        Muxes := 1
           2 Input      2 Bit        Muxes := 1
           7 Input      2 Bit        Muxes := 1
           5 Input      2 Bit        Muxes := 1
           2 Input      1 Bit        Muxes := 140
           4 Input      1 Bit        Muxes := 2
```

8.9.4.2 TRNG usage

Report

```
Detailed RTL Component Info :
+---XORs :
          16 Input      1 Bit         XORs := 8
```

8.9.4.3 KMU usage

Report

```
Detailed RTL Component Info :
+---Adders :
           2 Input     32 Bit       Adders := 1
           2 Input      8 Bit       Adders := 33
           2 Input      5 Bit       Adders := 7
           2 Input      4 Bit       Adders := 3
           2 Input      3 Bit       Adders := 1
           2 Input      2 Bit       Adders := 2
+---XORs :
           2 Input      4 Bit         XORs := 2
          16 Input      1 Bit         XORs := 8
           4 Input      1 Bit         XORs := 1
           2 Input      1 Bit         XORs := 22
           3 Input      1 Bit         XORs := 6
+---Registers :
                     152 Bit    Registers := 3
                     133 Bit    Registers := 3
                     128 Bit    Registers := 4
                      64 Bit    Registers := 1
                      15 Bit    Registers := 3
                       8 Bit    Registers := 33
                       7 Bit    Registers := 3
                       5 Bit    Registers := 6
                       4 Bit    Registers := 10
                       3 Bit    Registers := 1
                       2 Bit    Registers := 4
                       1 Bit    Registers := 96
+---Muxes :
           2 Input    152 Bit        Muxes := 6
           2 Input    133 Bit        Muxes := 7
           2 Input    128 Bit        Muxes := 7
           2 Input     64 Bit        Muxes := 2
           2 Input     16 Bit        Muxes := 48
           2 Input     15 Bit        Muxes := 10
           4 Input     15 Bit        Muxes := 1
           3 Input     15 Bit        Muxes := 1
           2 Input      8 Bit        Muxes := 50
           2 Input      7 Bit        Muxes := 2
           4 Input      7 Bit        Muxes := 1
           2 Input      5 Bit        Muxes := 27
```

```
  4 Input      5 Bit        Muxes := 1
  3 Input      5 Bit        Muxes := 1
  2 Input      4 Bit        Muxes := 9
  4 Input      4 Bit        Muxes := 1
  4 Input      3 Bit        Muxes := 1
  2 Input      2 Bit        Muxes := 4
  7 Input      2 Bit        Muxes := 1
  5 Input      2 Bit        Muxes := 1
  3 Input      2 Bit        Muxes := 1
  4 Input      2 Bit        Muxes := 2
 17 Input      2 Bit        Muxes := 1
  2 Input      1 Bit        Muxes := 174
  4 Input      1 Bit        Muxes := 15
```

8.10 Implementation results

The implementation phase has been done for 3 different modules: TRNG, PUF and the whole KMU. Also in this case the clock constraints have been modified in order to see the affections on power consumption and slack. In this phase both power optimization and place and routing optimization have been made. The number of LUTs involved is the one that is physically implemented in the NEXYS 4 FPGA.

8.10.1 PUF implementation

The approach is very similar to the one adopted for the synthesis phase. The results coming from the implementation are grouped in the following table. Starting from a clock constraints of 229 MHz several implementation have been performed decreasing the clock frequency.

clk[Hz]	Period[ns]	Power[mW]	Dyn[mW]	Stat[mW]	slack[ns]	SLICE LUTs	FF
$2.29 \cdot 10^8$	3	122	38	84	-0.360	658	549
$2.09 \cdot 10^8$	4	112	28	84	-0.318	651	549
$2.00 \cdot 10^8$	5	106	22	84	0.695	586	549
$1.67 \cdot 10^8$	6	102	18	84	1.933	585	549
$1.43 \cdot 10^8$	7	100	16	84	1.959	585	549
$1.25 \cdot 10^8$	8	98	14	84	3.793	585	549
$1.11 \cdot 10^8$	9	96	12	84	4.361	585	549
$1.00 \cdot 10^8$	10	95	11	84	5.222	585	549
$6.67 \cdot 10^7$	15	91	7	84	10.121	585	549
$5.00 \cdot 10^7$	20	89	5	84	15.683	585	549
$3.33 \cdot 10^7$	30	88	4	84	25.556	585	549
$2.50 \cdot 10^7$	40	87	3	84	33.829	585	549
$2.00 \cdot 10^7$	50	86	2	84	44.564	585	549

This results have been plotted to see the behavior of the unit and to better understand how the clock frequency affects the power, slack and LUTs number.

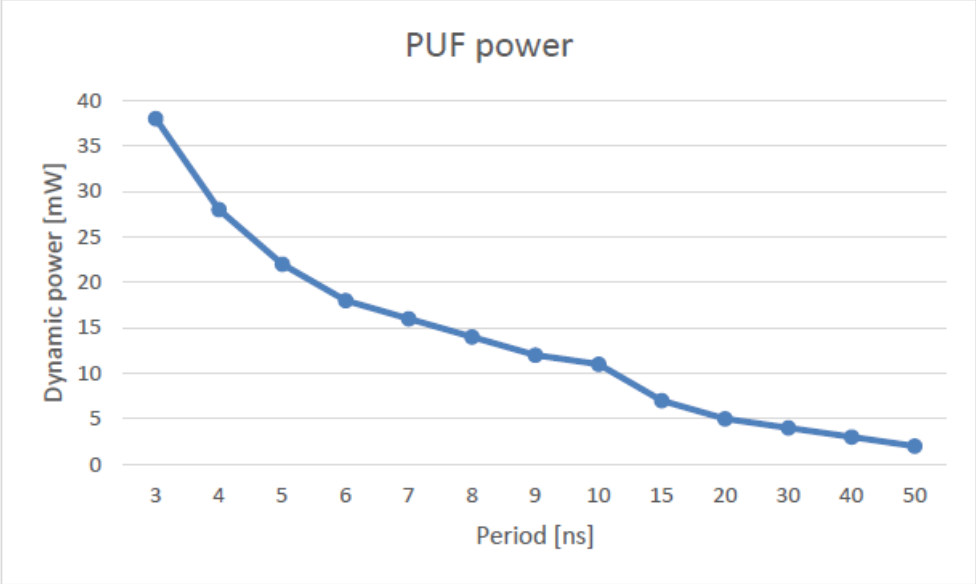

Figure 8.25: PUF dynamic power implementation

This first graph shows how the clock affects the PUF power consumption. Only the dynamic power have been plotted cause the static power is still the same (84 mW). As expected stricter constraints led to a higher power consumption and viceversa. Is interesting to notice how fast the dynamic power decreases with respect to the period.

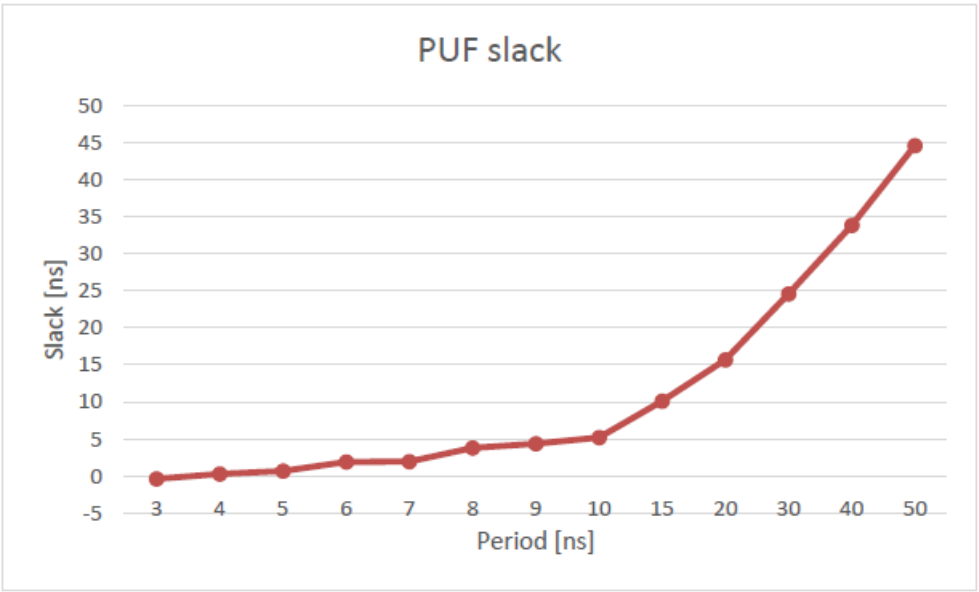

Figure 8.26: PUF slack implementation

The difference between the required time and the arrival time grows almost linearly. The slack become negative only when the period is shorter than 3 ns. The optimization phase has improved the maximum clock frequency of the device reaching 209 MHz.

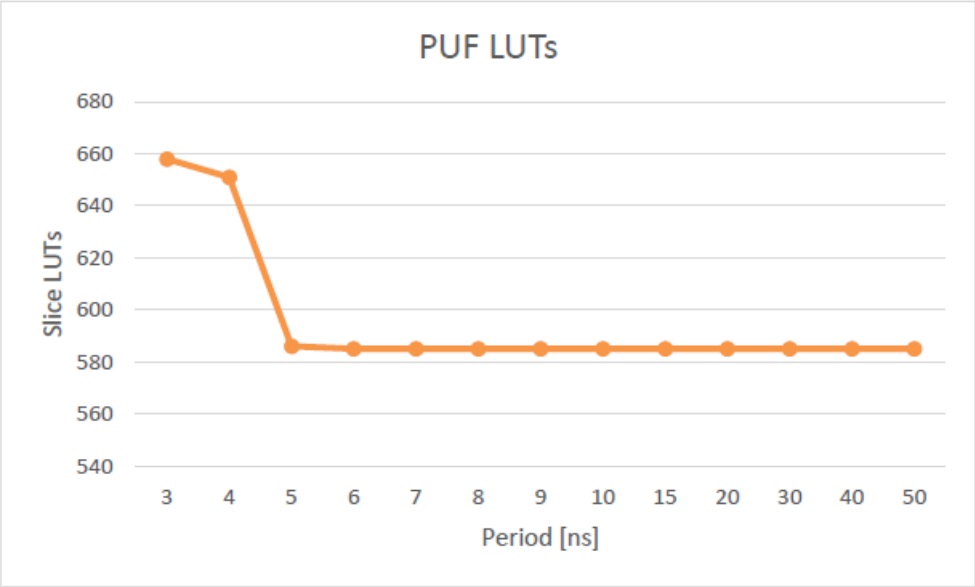

Figure 8.27: PUF LUTs implementation

The number of LUTS remains the same (585) for all the configuration except the the one with period equal to 4ns. As we can notice the unit is able to reach a higher operating frequency but with 76 more LUTs (area increasing).

8.10.2 TRNG implementation

Also in this case the results coming from the implementation phase are grouped in the following table. Inn order to get a negative slack the clock constraints started from 249 MHz down to 20 MHz.

clk[Hz]	Period[ns]	Power[mW]	Dyn[mW]	Stat[mW]	slack[ns]	SLICE LUTs	FF
$2.29 \cdot 10^8$	2	230	133	97	-0.002	290	136
$2.29 \cdot 10^8$	3	186	89	97	0.770	290	136
$2.09 \cdot 10^8$	4	164	67	97	1.521	290	136
$2.00 \cdot 10^8$	5	150	53	97	2.450	290	136
$1.67 \cdot 10^8$	6	141	44	97	3.458	290	136
$1.43 \cdot 10^8$	7	135	38	97	4.458	290	136
$1.25 \cdot 10^8$	8	130	33	97	5.130	290	136
$1.11 \cdot 10^8$	9	127	30	97	6.138	290	136
$1.00 \cdot 10^8$	10	124	27	97	7.064	290	136
$6.67 \cdot 10^7$	15	115	18	97	12.130	290	136
$5.00 \cdot 10^7$	20	110	13	97	17.138	290	136
$3.33 \cdot 10^7$	30	106	9	97	27.138	290	136
$2.50 \cdot 10^7$	40	104	7	97	37.138	290	136
$2.00 \cdot 10^7$	50	102	5	97	47.130	290	136

Also in this case this results have been plotted to see the behavior of the unit and to better understand how the clock frequency affects the power and slack. As we can notice the LUTs number is not affected by the clock constraint and it is always constant to the value 290.

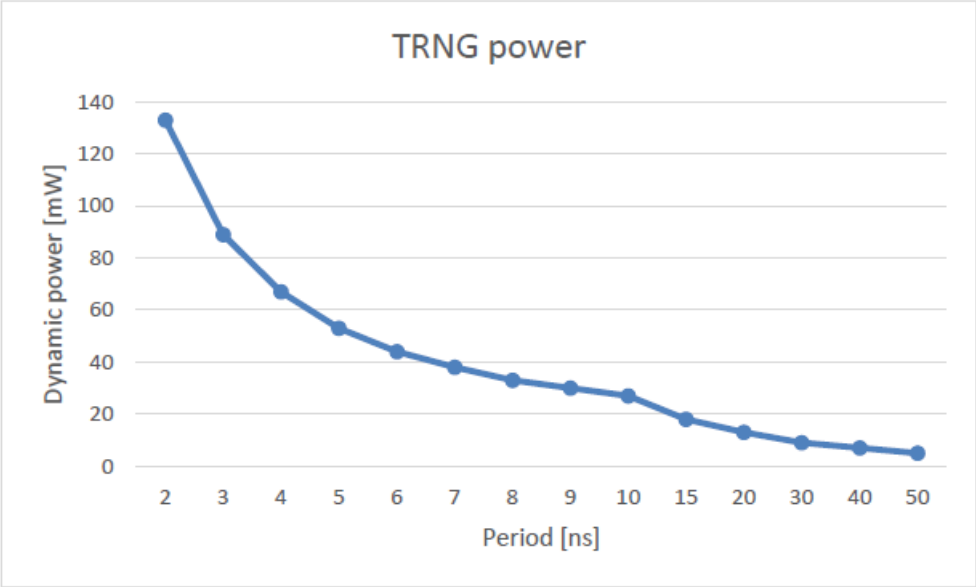

Figure 8.28: TRNG dynamic power implementation

This first graph shows how the clock affects the TRNG power consumption. Only the dynamic power have been plotted cause the static power is still the same (97 mW). As expected stricter constraints led to a higher power consumption and viceversa. Also in this case the dynamic power drops drastically with respect to the period.

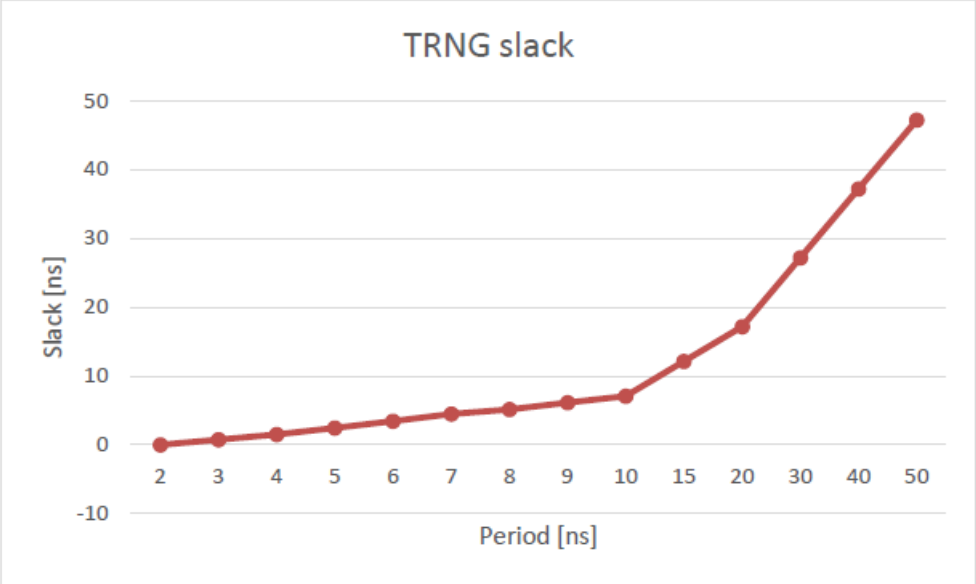

Figure 8.29: TRNG slack implementation

Also here the difference between the required time and the arrival time grows almost linearly. The slack become negative only when the period is shorter than 2 ns.

8.10.3 KMU implementation

These are the results coming from the implementation of the whole KMU.

clk[Hz]	Period[ns]	Power[mW]	Dyn[mW]	Stat[mW]	slack[ns]	SLICE LUTs	FF
$2.00 \cdot 10^8$	5	149	65	84	-0.839	3176	4235
$1.67 \cdot 10^8$	6	138	54	84	0.057	3175	4235
$1.43 \cdot 10^8$	7	126	42	84	0.047	3104	4235
$1.25 \cdot 10^8$	8	122	38	84	0.072	3119	4235
$1.11 \cdot 10^8$	9	116	33	84	0.354	3090	4235
$1.00 \cdot 10^8$	10	113	29	84	0.516	3086	4235
$6.67 \cdot 10^7$	15	104	20	84	1.900	3093	4235
$5.00 \cdot 10^7$	20	99	15	84	1.951	3105	4235
$3.33 \cdot 10^7$	30	94	10	84	6.697	3098	4235
$2.50 \cdot 10^7$	40	92	8	84	9.786	3105	4235
$2.00 \cdot 10^7$	50	90	6	84	14.441	3120	4235

This results have been plotted to see the behavior of the unit and to better understand how the clock frequency affects the power, slack and LUTs number.

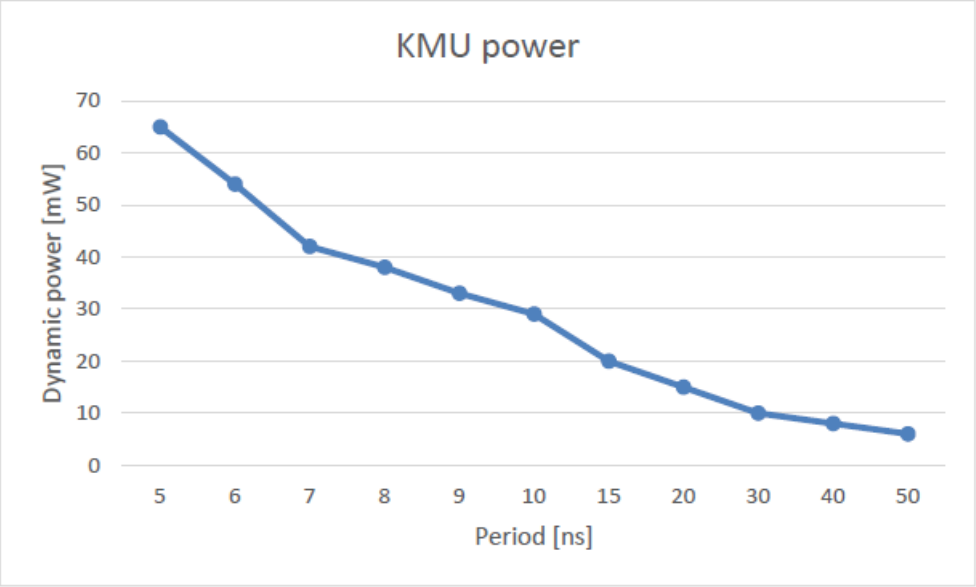

Figure 8.30: KMU dynamic power implementation

This first graph shows how the clock affects the KMU power consumption. Also in this case only the dynamic power have been plotted cause the static power is still the same (84 mW). As expected stricter constraints led to a higher power consumption and viceversa. Also here is interesting to notice how fast the dynamic power decreases with respect to the period. In particular for a period equal to 50 ns, the dynamic power consumption is around 9 times less with respect to a period of 6 ns.

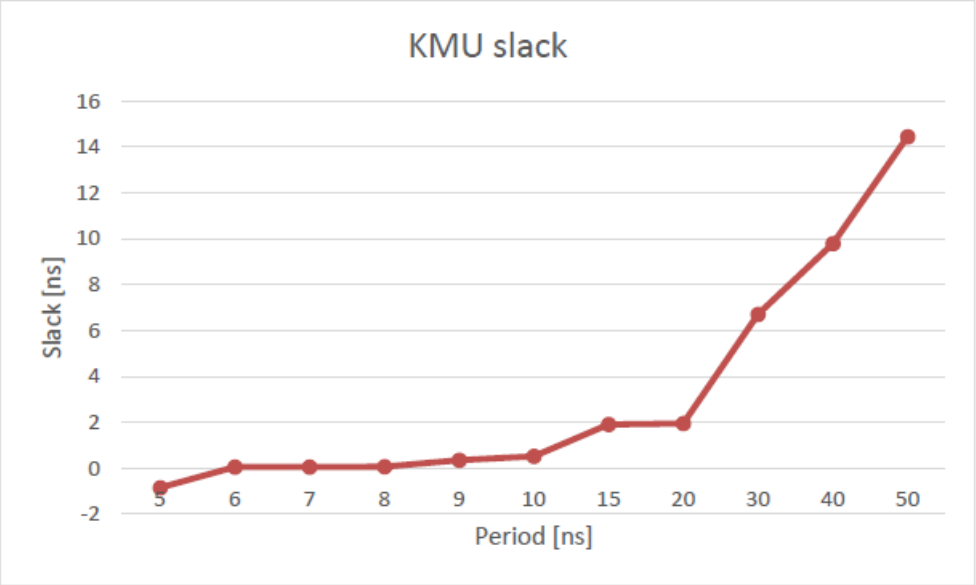

Figure 8.31: KMU slack implementation

The difference between the required time and the arrival time does not grow very linearly. The slack become negative only when the period is shorter than 5 ns (200 MHz). We have similar values of slack when the period is set to 6, 7 and 8 ns respectively. What changes in these 3 configurations are the dynamic power consumption and the number of LUTs used. Here is where the power consumption optimization and place and routing optimization leads to have the greatest benefits.

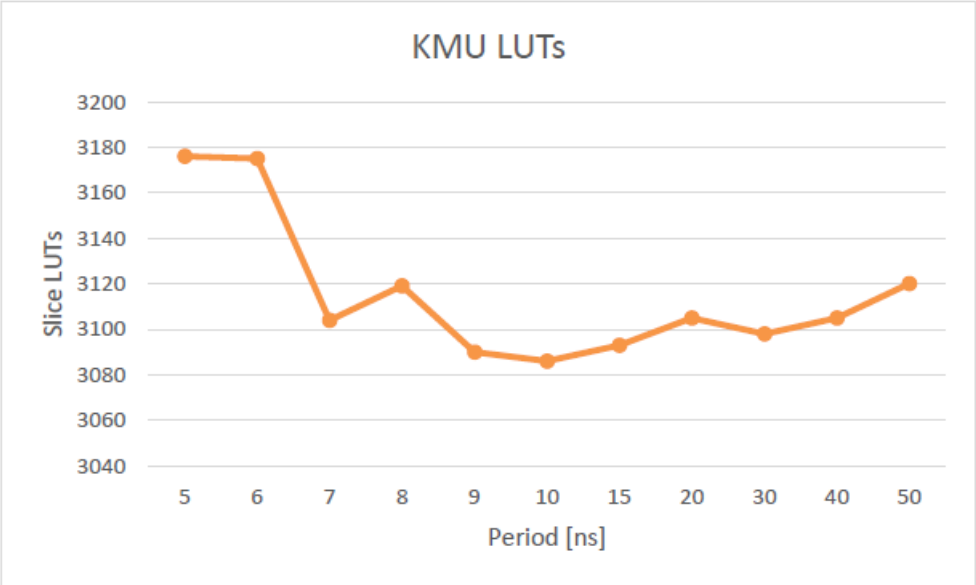

Figure 8.32: KMU LUT implementation

The behavior here is not linear. For every clock constraint the place and routing optimization tries to do the best best placement. As result it may happens that higher operational frequency is associated to a lower number of LUTs and viceversa. The best achieved result is with a clock constraint of 10 ns with a number of used LUTs equal to 3086.

8.11 AXI interface

The ARM Advanced Microcontroller Bus Architecture (AMBA) is an open standard, mainly used in SoC design. It is able to manage connection and specification of functional blocks.
The main features of this protocol are:

- data phases and address/control are separated
- byte strobes are used to support unaligned data transfer
- low-cost Direct Memory Access (DMA) is possible thanks to two separated data-channel (read and write)
- out-of-order transaction completion

A lot of benefits derives from using AMBA, it enables efficient IP re-use, is suitable for low-power operations and offers a wide flexibility to work with very different SoCs.

8.11.1 Architecture

AXI is a burst-based protocol transaction. Write and read data channels are separated, one of each goes from the master and the slave (write) and viceversa (read). During the write operation, the slave must be able to communicate the end of the write transaction. In order to do this an additional response is added.

The following figures shows the behavior of read and write operations.

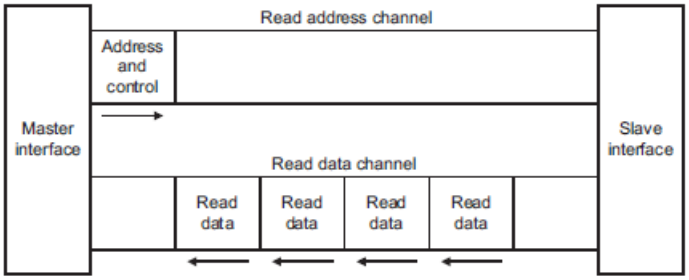

Figure 8.33: read transaction

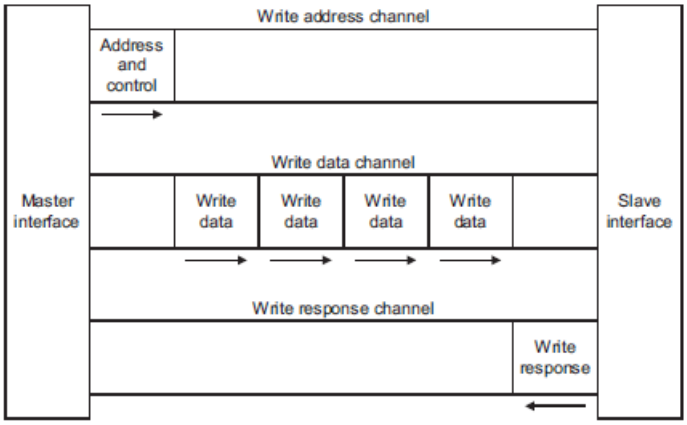

Figure 8.34: write transaction

8.11.2 Interface and Interconnect

The Figure 8.35 shows a typical example of this architecture, a consistent number of masters and slaves are interconnected through their interfaces.

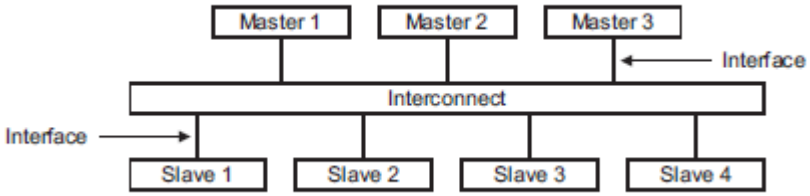

Figure 8.35: Interconnection

There is a single interface definition for the different kind of interconnections (master and slave, master and intercon., slave and intercon.)

CHAPTER 9

ROCKET-CHIP

System-on-chip (SoC) is able to boost efficiency by using integration and customization. It is an open source technology, that can be obtained under a BDS licence on Github, developed at UC Berkeley for both industrial and research aims. Rocket Chip is not a single instance of an SoC design. On the contrary it is a design generator that allows to produce from a single source a lot of different design instances. Thanks to the high numbers of parameters that can be set it can be customized for any particular application. With the aim of enlarge its modularity a lot of component libraries are provided as independent repositories.

9.1 Background

Starting from RISC-V Instruction Set Architecture (ISA) researchers at UC Berkeley have developed the Rocket Chip. One of his strong point is the fact that it is open and free meaning that it can be easily used in all the environments, from open-source to commercial.
Thanks to its modular design RISC-V is extremely flexible. This modularity is represented by the existence of approximately 40 integer instructions that must be implemented by all cores and by the wide opcode space that has been left over in order to support optional extensions, even if, among these optional extensions, the most conventional have already been standardised. Multiply and divide, atomics, single-precision and double-precision floating point are existing extensions called common extensions (IMAFD) and are enclosed into the extension that provides a general-purpose, scalar instruction set.

There are different options regarding the address mode (32, 64 and 128 bits plus 16-bit compressed extension used to reduce static code size). Non-standard extensions are located in the opcode space instead. This division has been made in order to allow designers to add new features to processors without generating a conflict between them and existing compiled software.
Chisel is an open-source hardware construction language embedded in Scala and it is used to implement Rocket Chip. Chisel is directly responsible of the circuits and, since it is part of Scala, it makes available its full programming language to generate circuits, provide functional and object-oriented descriptions of circuits. Moreover, the advantage of Chisel is the availability of a higher number of features with respect to those found in Verilog. By using Chisel, it is possible to generate both a synthesizable Verilog code and a RTL simulator implemented in C++. The two are equivalent but while the first is compatible FPGA and ASIC design tools the second is significantly faster and a complete Rocket Chip instance can be simulated using it.

9.2 Rocket Chip Generator

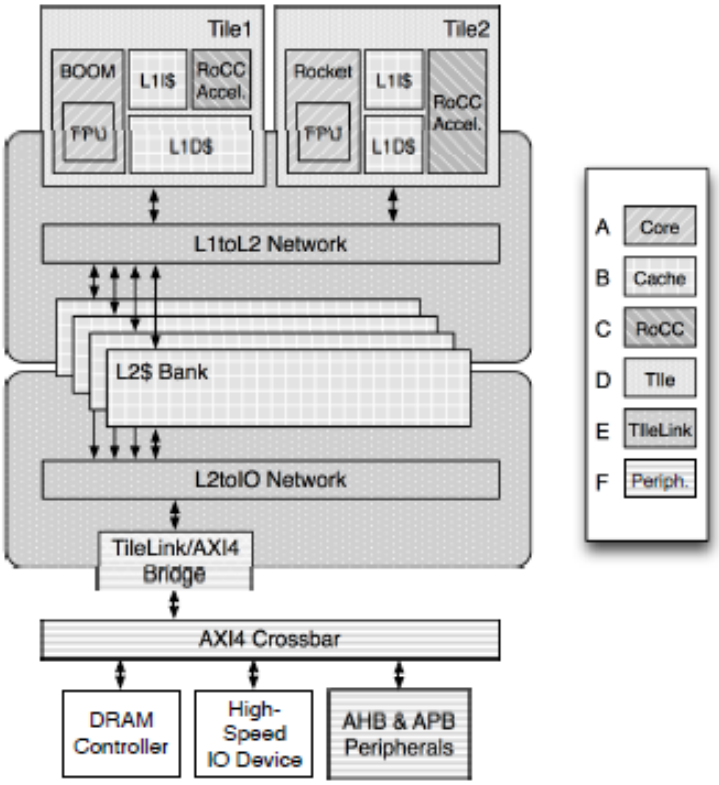

Figure 9.1: The Rocket Chip instance

The Rocket Chip generator is made of several sub-components generator (core, cache, RoCC-compatible coprocessor, Tile, TileLink) and some peripherals.

The programming language Chisel and the RISC-V-based platform represent the basis for the development of the Rocket Chip generator. This generator includes a collection of parametrized chip-building libraries that allow to produce different variations of the SoC.
Figure 9.1 represents a Rocket Chip. It is possible to notice that 2 tiles are connected to a 4-bank L2 cache. An AXI interface is responsible for the interconnection between the memory system and the external I/O.

The aim of the Rocket Chip is to support a different variety of workloads and to enhance energy efficiency, for these purposes, heterogeneity is supported. Indeed, it can be made of different tiles and, more important, it can even support the addiction of custom accelerators. In order to do this, it is able to support three different mechanisms. Which one to choose depends from the extent of the involvement between the accelerator and the core. The easiest option is also the one referring to the most tightly coupled option.
If the purpose is more decoupling it is then necessary to make the accelerator work as a coprocessor so thanks to the RoCC interface the processor can send to it data and commands. When the aim is fully decoupled accelerators it is possible to instantiate them in their own tiles and then to connect them, in a coherent way, using TileLink. It is possible to combine these techniques, and so interim solutions can be reached.

CHAPTER 10

CONCLUSIONS

In this thesis document a new Key Management Unit for RISC-V secure processor is presented. This approach has been tested on NEXYS 4 FPGA and is now under testing on Kyntex FPGA. Results showed savings up to 89% dynamic power in Artix-7 under different constraints. We can also notice that the optimal placement in terms of area is obtained with a clock period equal to 10 ns. The new adopted PUF solution has been tested on an Artix-7 Nexys 4 FPGA board and results inuniqueness of 49.2%, uniformity of 48.5% and 47.8% bit-aliasing. The noisy bits are recoveredusing BCH error correction scheme that allow to obtain final response with a reliability of 99.99%.

Future work is the implementation of this unit with the rocket chip whit a full immersive testing. The same team with which I worked is now doing this operation.

BIBLIOGRAPHY

[1] ARM, *AMBA AXI Protocol v1.0*, 2004.

[2] K. ASANOVIĆ, R. AVIZIENIS, J. BACHRACH, S. BEAMER, D. BIANCOLIN, C. CELIO, H. COOK, D. DABBELT, J. HAUSER, A. IZRAELEVITZ, S. KARANDIKAR, B. KELLER, D. KIM, J. KOENIG, Y. LEE, E. LOVE, M. MAAS, A. MAGYAR, H. MAO, M. MORETO, A. OU, D. A. PATTERSON, B. RICHARDS, C. SCHMIDT, S. TWIGG, H. VO, AND A. WATERMAN, *The rocket chip generator*, Tech. Rep. UCB/EECS-2016-17, EECS Department, University of California, Berkeley, Apr 2016.

[3] W. J. M. B. SUNAR AND D. R. STINSON, *A provably secure true random number generator with built-in tolerance to active attacks*, vol. 56, IEEE Transactions on Computers, 2007.

[4] A. BRADBURY, G. FERRIS, AND R. MULLINS, *Tagged memory and minion cores in the lowrisc soc*, tech. rep., lowRISC project team - Computer Laboratory - University of Cambridge, Dec 2014.

[5] B. GASSEND, D. CLARKE, M. V. DIJK, AND S. DEVADAS, *Controlled physical random functions*, 18th Annual Computer Security Applications Conference, 2002.

[6] B. GASSEND, D. DWAINE, M. VAN MARTEN, , AND S. DEVADAS, *Silicon physical random functions*, Proceedings of the 9th ACM Conference on Computer and Communications Security, 2002.

[7] C. GU, N. HANLEY, AND M. O'NEILL, *Improved reliability of fpga based puf identification generator design*, vol. 10, ACM Trans. Reconfigurable Technol. Syst., 2017.

[8] Y. HORI, H. KANG, T. KATASHITA, AND A. SATOH, *Pseudo-LFSR PUF : A Compact, Efficient and Reliable Physical Unclonable Function*, International Conference on Reconfigurable Computing and FPGAs, 2011.

[9] B. JUN AND P. KOCHER, *The Intel Random Number Generator*, White paper prepared for Intel Corporation, 1999.

[10] KNUTWOLD AND C. H. TAN, *Analysis and Enhancement of Random Number Generator in FPGA Based on Oscillator Rings*, International Journal of Reconfigurable Computing, 2009.

[11] S. S. KUMAR, J. GUAJARDO, R. MAES, G. J. SCHRIJEN, AND P. TUYLS, *The butterfly puf protecting ip on every fpga*, IEEE International Workshop on Hardware-Oriented Security and Trust, 2008.

[12] J. W. LEE, D. LIM, B. GASSEND, G. E. SUH, M. V. DIJK, AND S. DEVADAS, *A technique to build a secret key in integrated circuits for identification and authentication applications*, Symposium on VLSI Circuits. Digest of Technical Papers (IEEE Cat. No.04CH37525), 2004.

[13] D. LIM, J. W. LEEAND, B. GASSEND, G. E. SUH, M. V. DIJK, AND S. DEVADAS, *Extracting secret keys from integrated circuits*, vol. 13, IEEE Trans. Very Large Scale Integr. Syst., 2005.

[14] R. MAES, A. V. HERREWEGE, AND I. VERBAUWHEDE, *Pufky: A fully functional puf-based cryptographic key generator*, Cryptographic Hardware and Embedded Systems – CHES 2012, 2012.

[15] R. PAPPU, B. RECHT, J. TAYLOR, AND N. GERSHENFELD, *Physical one-way functions*, vol. 297, 2001.

[16] M. R. RAO AND R. K. SHARMA, *FPGA implementation of combined AES-128*, 8th International Conference on Computing, Communication and Networking Technologies (ICCCNT), 2017.

[17] G. E. SUH AND S. DEVADAS, *Physical unclonable functions for device authentication and secret key generation*, Proceedings of the 44th Annual Design Automation Conference, 2007.

[18] M. M. WONG, J. HAJ-YAHYA, S. SAU, AND A. CHATTOPADHYAY, *A New High Throughput and Area Efficient SHA-3 Implementation*, IEEE International Symposium on Circuits and Systems (ISCAS), 2018.

[19] D. YAMAMOTO, K. SAKIYAMA, M. IWAMOTO, K. OHTA, T. OCHIAI, M. TAKENAKA, AND K. ITOH, *Uniqueness enhancement of puf responses based on the locations of random outputting rs latches*, Proceedings of the 13th International Conference on Cryptographic Hardware and Embedded Systems, 2011.