DEPARTMENT OF ELECTRICAL ENGINEERING
INDIAN INSTITUTE OF TECHNOLOGY MADRAS
CHENNAI - 600036

IMPLEMENTATION OF A 128-BIT ARBITER PUF ON A FPGA

A Project Report

Submitted by

VADLA RAMA KISHAN

(EE19M054)

In the partial fulfilment of requirements

For the award of the degree

Of

MASTER OF TECHNOLOGY

JUNE, 2021

CERTIFICATE

This is to undertake that the Project report titled **IMPLEMENTATION OF A 128-BIT ARBITER PUF ON A FPGA**, submitted by me to the Indian Institute of Technology, Madras for the award of the degree in M.Tech, is a bona fide record of the research work done by me under the supervision of Prof. V. Kamakoti. In whole or in parts, the contents of this Project report have not been submitted to any other Institute or University for the award of any degree or diploma.

Place: Chennai,600036

Date: June 2021

VADLA RAMA KISHAN

EE19M054

Prof.V. Kamakoti

Project Guide

Prof. Anbarasu M

Project Co-Guide

ACKNOWLEDGEMENTS

I wish to express my wholehearted gratitude to my project guide **Dr. V. Kamakoti**, Professor in the Department of Computer Science and Engineering, for his valuable advice and technical support from the beginning of my project.

I owe a particular debt of gratitude to **Dr. Anbarasu M**, Associate Professor in the Department of Electrical Engineering for being my co-guide.

Foremost, I would like to express my sincere gratitude to **Arjun Menon** for providing all facilities, timely suggestions and guidance to do this project.

My heartiest thanks to the friends who have supported me in all possible ways. Words are inadequate to express my gratitude to my family and friends who have supported me at all times. I would also like to thank my institution and the faculty members without whom this project would have been a distant reality. Above all i bow before the Almighty for his immense blessings throughout our lives.

Thank you all

Vadla Rama Kishan

EE19M054

ABSTRACT

Keywords: FPGA: Field Programmable Gate Array, IOT: Internet Of Things.

This is the age of Digital Revolution. Today devices are not only connected by wires. They are connected wirelessly forming a network through the use of internet which can otherwise be called as Internet of Things. Internet of Things (IOT) is diffused so much in the fields smart home, transportation, medical and health care, agriculture, energy management, environmental monitoring, military applications etc. which affect our daily lives in one or the other way. As these devices are collecting, analysing our data, they are prone to various types of attacks. In brief many of the devices in Internet of Things are depended on peer to peer authentication where the information is stored in non-volatile memories which becomes the vulnerable spot for attacks. This lead to usage of a new primitive i.e. a PUF which generates and stores secret keys to authenticate a device. A PUF is a device where given a challenge as input produces a response as a output which is unique to that particular device. This work explains various types of PUFs, parameters to analyse PUFs and it mainly deals with Arbiter PUF: working, implementation on FPGA.

TABLE OF CONTENTS

LIST OF TABLES

Table	Title	Page

LIST OF FIGURES

ABBREVIATIONS

PUF expanded form is PHYSICALLY UNCLONABLE FUNCTION

MOS expanded form is METAL OXIDE SEMICONDUCTOR

FPGA expanded form is FIELD PROGRAMMABLE GATE ARRAY

ASIC expanded form is APPLICATION SPECIFIC INTEGRATED CIRCUIT

API expanded form is APPLICATION PROGRAMMING INTERFACE

IOT expanded form is INTERNET OF THINGS

NVM expanded form is NON VOLATILE MEMORY

SRAM expanded form is STATIC RANDOM ACCESS MEMORY

RO expanded form is RING OSCILLATOR

CRP expanded form is CHALLENGE RESPONSE PAIR

ICs expanded form is INTEGRATED CIRCUITS

LFSR expanded form is LINEAR FEEDBACK SHIFT REGISTER

VIO expanded form is VIRTUAL INPUT OUTPUT

GUI expanded form is GRAPHICAL USER INTERFACE

CLB expanded form is CONFIGURABLE LOGIC BLOCK

LUT expanded form is LOOK UP TABLE

CHAPTER 1

INTRODUCTION AND MOTIVATION

1.1 INTRODUCTION:

Now a days, smart devices has grown to the point that they are profoundly embedded in people's lives. These smart devices are connected by building an Internet of Things (IoT) network [Ashton *et al.* (2009)]. As smart devices making our daily lives effortless, there lies so many challenges such as efficiency, power consumption, performance, security etc. Security of the smart devices is the most important as they are exposed to API (Application Programming Interface) attacks, viruses, malware easily and are hard to protect. Thus hardware security is of prime importance today. Smart devices interact with each other by peer to peer identification and authentication. This is done by storing secrets in non-volatile memory (NVM). Storing secrets in NVM works well only in secured environments. Malicious parties use the stored information in NVM for identity spoofing. As a result, physically unclonable function (PUF) was created. PUF is a device that, when given a challenge, generates a response that is unique to that device as an output. They are primitives which were introduced in [Pappu *et al.* (2002*a*)], extracts unique information from inbuilt variations like load capacitance, MOS doping present in physical properties of the device or a chip. Though the layout masks are same, the delay between ICs are unique because of the manufacturing process variations that incurred to them. They are both secured from malicious attacks and also are cost-effective. PUFs can also be used to safeguard intellectual property (IP), activate remote services, and store secret keys [Guajardo *et al.* (2008)]. Silicon PUFs [Gassend *et al.* (2002*a*)] works by exploiting the intrinsic delay that are formed due to imperfections in ASICs and FPGAs during fabrication. This helps PUF to generate secret keys. Thus we use PUF for authenticating devices and fingerprinting instead of storing secrets in NVM. There are so many different types of PUFS like RO PUF, Butterfly PUF, Arbiter PUF, Glitch PUF, SRAM PUF which are either enhanced version of previous

PUFs or one of the many ways to generate challenge response pairs i.e. CRPs, and one of the many classifications of PUFs like PUFs that are based on storage and PUFs that are based on delay and sub classes like Strong PUFs and weak PUFs. For example, Arbiter PUF which is a strong PUF was proposed which uses the delays that are caused by the imperfections in identically laid-out delay paths [Lim *et al.* (2005)]. One more differentiation that can be used to distinguish PUFs is based on amount of CRPs. They are differentiated as strong PUFs and weak PUFs. With increase in challenge bit size, the amount of CRPs of the weak PUF increases linearly and in a strong PUF it increases in the power of 2. For example in a 128 bit arbiter PUF the maximum number of CRPs that are possible is 2^{128}. The generated output of the PUF which is called as response should have low intra hamming distance and high inter hamming distance for successful identification and authentication. To store this we need a large database of memory. Previously this used to be done by using client-server based model where client used to be an instantiation of PUF and server used to store the CRP table, that way authentication use to happen among devices. In the later works the CRP table was stored in device itself which provided some resistance to attacks. In this work it is implemented in such a way that the challenges are generated on the spot randomly, so that the device only stores responses there by being more resistant to malicious attacks. The PUF that is implemented is arbiter PUF.

1.2 MOTIVATION:

Electronic devices becoming a crucial part of our life is increasing day-by-day. This has raised concerns about device security, which is particularly critical for military systems. Safety of the device is the number 1 parameter that is very important. Cryptography which rely on the idea of the binary key comes into the picture which provides several measures to this problems. As reality always differs from ideal nature , securing the device from all these physical attacks where side channel attacks is one of the many examples and software attacks where viruses, API attacks are the examples that results in full security breaks [Rührmair *et al.* (2013)]. Previously it used to be a client-server based model where NVM which is abbreviated as non-volatile memory stores all the secrets. Since this does not resulted in a better hardware security, this lead to the introduction of PUFs where there will be a prover device and verifier device to perform authentication protocol. But even the PUF which stores CRP table in the verifier device is not resistant to attacks.

This is the motivation that lead to the development of a PUF where only responses are stored in the verifier device and challenges are generated randomly through the use of a LFSR which is abbreviated as linear feedback shift register and placement of muxes are done manually which gives a symmetrical layout of the design thereby optimising routing.

CHAPTER 2

RELATED WORK

From the first introduction of PUF in 2002 by [Pappu *et al.* (2002*b*)], there were many PUFs that were introduced targeting FPGAs and ASICs according to the requirements for the optimisation in power, performance and the security of electronic devices and for the efficient control of information over the network. Many researchers have introduced a wide variety of PUFs like SRAM PUF [Guajardo *et al.* (2007)], Arbiter PUF [Gassend *et al.* (2002*b*)][Gu *et al.* (2016)], Latch PUF [Su *et al.* (2008)], Bistable Ring PUF [Chen *et al.* (2011)], Flip-flop PUF [Maes *et al.* (2008)], RO PUF which is abbreviated as ring oscillator PUF, Reconfigurable PUF [Kursawe *et al.* (2009)], Configurable RO PUF [Yu *et al.* (2011)], Buskeeper PUF [Simons *et al.* (2012)], processor-based PUF [Maiti and Schaumont (2012)], and Butterfly PUF [Kumar *et al.* (2008)].

The Arbiter PUF works by the delay concept taking the difference of arrival times of the both signals. If the first signal arrives first then the output is taken as 1 and vice versa. The RO PUF works by taking between two identical ring oscillators for which outputs are two incrementing counters and the output values from counters are compared using a comparator. SRAM PUF manipulates the starting state of SRAM cells in distinct memory blocks on multiple FPGAs to generate IDs. Despite the fact that FPGAs feature SRAM memory, and some feature a pre-set initial state that prohibits them from booting or starting off with a random value. As a solution to this problem, Butterfly PUF is like an extended version of SRAM PUF that can be activated at whatever the time we want, unlike SRAM PUF, which only can be activated at power-up of the device [Gu *et al.* (2017)]. The PUF according to designs can be distinguished as strong PUF and weak PUF. In a strong PUF, with increase of bit size in challenge the number of CRPs are raised with the power of 2 and in weak PUF increase linearly with increase of bit size in challenge.

CHAPTER 3

DESCRIPTION OF PUF

3.1 INTRODUCTION OF PUF

PUFs which are abbreviated as physically unclonable functions are a very few of the most promising solutions in the security of the hardware. They could be designed, implemented and authenticated in both ASICs and FPGAs. From the name itself through the use of unclonable property, a PUF cannot be reproduced. A PUF helps the smart device that it got embedded in by making the device uniquely identifiable. The properties of uniqueness and unclonable make a PUF as one of its kind in the field of hardware security. These properties are attained in a PUF due to inbuilt variations like load capacitance, MOS doping present in physical properties of the device or a chip while fabrication. The word imperfection which is generally a negative word works positively in case of the PUF by making it not replicable. PUFs are tamper-evident equipments, as any attempt of harmful alterations on the PUFs are easily detected.

A PUF is a physical unit that works in the same way as a function, transferring a set of challenges to a set of responses resulting in a set of CRPs which are abbreviated as challenge-response pairs that cannot be cloned and one of a kind for any smart device where the PUF is used. [Barbareschi *et al.* (2015)].

3.2 DIFFERENT TYPES OF PUF

Many researchers have developed various types of PUFs according to the requirement and application. PUFs can be split into different types of subclasses where each may have one or more subclasses attached to them.

PUFs, taking the physical randomness into account can be divided into 2 types:

1 **Physical randomness explicitly introduced PUFs:**
In this randomness is introduced from the outside. It is less affected to environmental variations and has a far more greater ability in differentiating smart devices when compared to (b). This is due to control of the randomness and other parameters with the designer/user.

2 **Physical randomness intrinsically present PUFs:**
The intrinsic randomness arises due to imperfections that occurred during fabrication. This type of randomness is inherently present in the devices. This type of devices are easier to construct when compared to 1.

Physical randomness intrinsically present PUFs are split into two categories:

A **PUFs depending on the delay:**

B **PUFs depending on the memory:**

A **PUFs depending on the delay:**
These PUFs generate responses from the hidden delay information of ICs. These are the PUFs that are operated due to the delay caused by imperfections in the fabrication process. Since this type of PUFs are measuring delay, they need additional hardware when compared to Memory based PUFs. Examples are arbiter PUF, RO PUF, xor PUF,butterfly PUF.

B **PUFs depending on the memory:**
These PUFs generate responses from the fixed pre state values present in the memory that caused during fabrication. These are PUFs that are operated using the power-up values of cells of SRAM that are present or flip-flops, which are naturally present in any IC device. SRAM PUF and flip flop PUF are some of the examples.

PUFs according to the number of challenges and applications are split into 3 categories:

a **Strong PUF**

b **Weak PUF**

c **Controlled PUF**

a **Strong PUF:**
These are the type of PUFs which have a complex challenge-response relation. With the raise in the hardware the maximum amount of challenges increases exponentially. These are the PUFs that are used for authentication. Examples are arbiter PUF, xor PUF.

b **Weak PUF:**
The PUFs that have a simple challenge response relation come under this category. With the raise in the hardware the maximum amount of challenges increases linearly. These are the PUFs that are used for the generation of secret keys and fingerprinting of silicon devices. Example is RO PUF, SRAM PUF.

c **Controlled PUF:**
These are the third type of PUFs which are made up of on the backbone of a strong PUF and a control logic that protects it. We can assume as a small box inside a big box. Here the small box is strong PUF and the big box is controlled PUF. Here, whatever has to be done, it has to be done through the big box. The inputs and outputs of the small box i.e. the strong PUF's input which is a challenge and output which is response cannot be directly accessed, but the logic named as one way hash function protects it from getting accessed. The input given to the big box i.e. The hash function pre-processes the challenges given at the input of controlled PUF before they the Strong PUF's input i.e. to reach to the input of the small box and the Strong PUF's output which are responses i.e. the outputs of small box are post processed by the logic i.e. one way hash function before the Controlled PUF lets them come from output. This preprocessing & postprocessing steps gives more security from the modelling attacks [6].

Down are some of the PUFs with the respective year of when they were first introduced.

- IC IDENTIFICATION USING DEVICE MISMATCH was introduced in 2000.
- PHYSICAL ONEWAY FUNCTION was introduced in 2001.
- PHYSICAL RANDOM FUNCTION was introduced in 2002.
- ARBITER PUF was introduced in 2004.
- COATING PUF was introduced in 2006.
- RO PUF AND SRAM PUF was introduced in 2007.
- BUTTERFLY PUF was introduced in 2008.
- PUF USING POWER DISTRIBUTION SYSTEM OF AN IC was introduced in 2009.
- GLITCH PUF was introduced in 2010.
- MECCA PUF was introduced in 2011.

3.3 BRIEF DESCRIPTIONS OF SOME PUFS

3.3.1 SRAM PUF

To distinguish devices such as micro controllers from any other, it is based on the fundamental characteristic of conventional SRAM memory, which is present in any electronic device. Each SRAM cell has its own preferred state when the SRAM is powered, which is determined by random variances in threshold voltages. The starting values of SRAM memory are determined by each SRAM cell's own preferred state, which is produced by randomization. Thus a SRAM response becomes unique which gives output of random values of 0's and 1's which is nothing but a fingerprint of chip. This pattern of random values or what we call fingerprint is not easily unclonable and unique to a particular SRAM which means to a particular chip. The figure of SRAM is shown in the Figure 3.1.

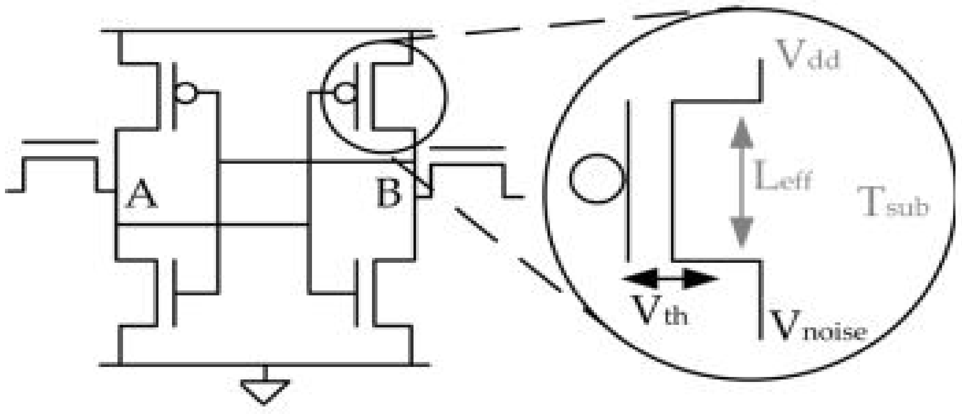


Fig. 3.1: SRAM cell with process variation and noise

This unique fingerprint becomes the base of hardware security. But the problem with this is that it contains noise. To remove noise and extract secret key, processing of fingerprint needs to be done. In this way since the key is not stored, hence becomes a safest way even when attacked. This way of extracting a secret key is more secured than the conventional way of storing secret parameters in NVM [22].The figure of SRAM PUF working is shown in the Figure 3.2.

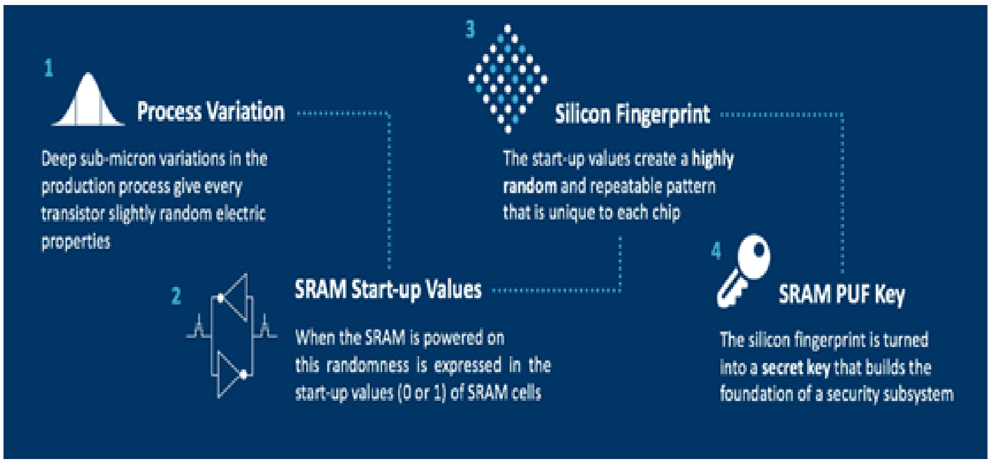

1
Process Variation
Deep sub-micron variations in the production process give every transistor slightly random electric properties
2
SRAM Start-up Values
When the SRAM is powered on this randomness is expressed in the start-up values (0 or 1) of SRAM cells
3
Silicon Fingerprint
The start-up values create a highly random and repeatable pattern that is unique to each chip
4
SRAM PUF Key
The silicon fingerprint is turned into a secret key that builds the foundation of a security subsystem

Fig. 3.2: Working of SRAM PUF

BUTTERFLY PUF

The problem with the SRAM PUF is that it is not supported on all FPGAs because of the uninitialized state of SRAM cell. So, in the FPGAs where this is not supported, the state is initialized to a known state.

This lead to introduction of butterfly PUF which is supported on all FPGAs. Butterfly PUF works on the concept of cross coupled circuits.

Cross coupled circuits is a fundamental design component for latches, flip flops and SRAM that is used to construct or create them. It's built in such a way that positive feedback is used to store the needed bit. A figure of a cross coupled inverter is shown below in Figure 3.3.

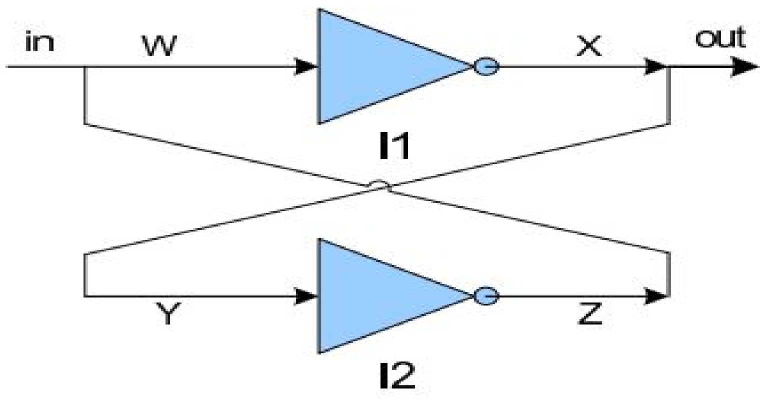


Fig. 3.3: Cross Coupled Inverter

BUTTERFLY PUF

We can observe from the Figure 3.4. that cross coupled inverter has one unstable state and two stable states. Stable states are used to store values. We can transition from an unstable condition to a stable or steady condition by applying an signal from outside to the input or by altering the elements utilised to construct the circuit. This concept is used to build butterfly PUF.

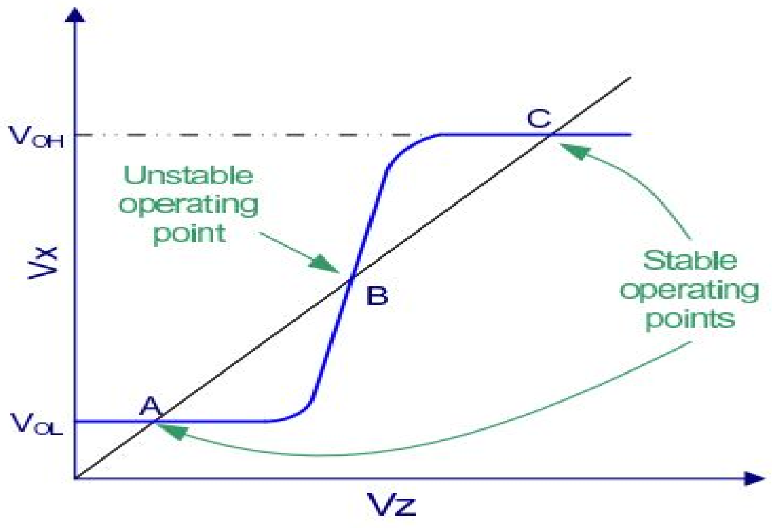


Fig. 3.4: States of Cross Coupled Inverter

BUTTERFLY PUF is designed such that during the starting phase, the notion is built on establishing a circuit that acts similarly to an SRAM cell. Butterfly PUF requires layout symmetry.

When the operation starts, the signal at the input is turned high. The Butterfly PUF circuit becomes unstable as a result of this. The excite signal is turned to low after a few clock cycles. As a result, the PUF enters one of the two stable or steady states. By fluctuations in the connecting wire delays the stable state is determined, which are produced by intrinsic IC features. When an attack happens, the attacker cannot know the stable states from the bit stream since it does not contain these values and also an attacker cannot know from chip because of the minute physical property variations in the elements [19]. The structure of butterfly PUF is shown in the Figure 3.5.

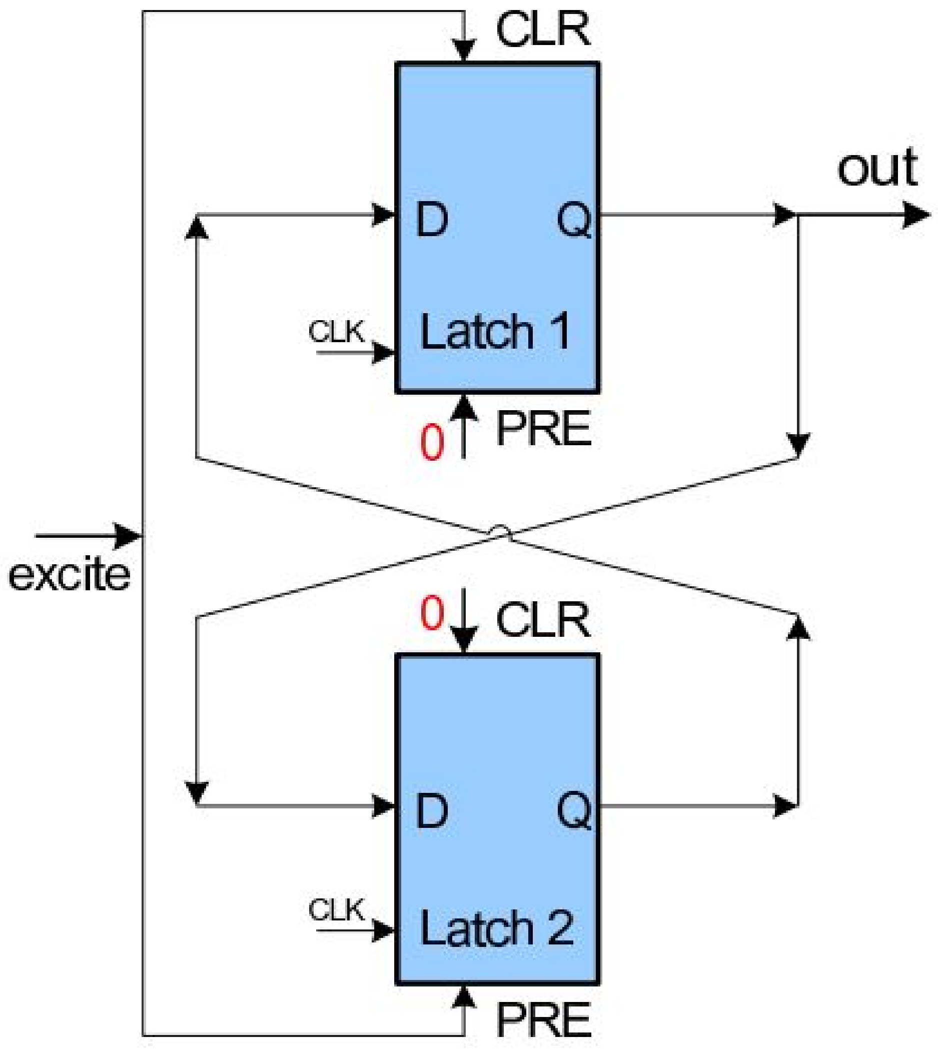


Fig. 3.5: Structure of a Butterfly PUF

3.3.2 RO PUF

All the elements may not have an inbuilt memory to implement memory based PUFs. The solution for that problem comes from the PUFs which are delay based where the delay is because of the intrinsic process variations that were inherently present in the device.

The RO PUF (Ring Oscillator PUF) belongs to the delay based PUFs group. A ring oscillator-based PUF looks to be a potential option. Every challenge supplied at the input generates an n-bit response from n ROs. The goal is to obtain a set of p bits from a total of n bits that can be used as a hardware fingerprint. A good response should have a small intra-Hamming distance but a large enough inter-Hamming distance to facilitate identification.

MOS doping, load capacitance, and line propagation are the properties that contribute to the unpredictability of a ring oscillator. This means that each RO has its own frequency. A counter is attached to the output of a RO. Through this we can measure frequency and when compared with another RO we extract 0 or 1 [Kodỳtek and Lórencz (2015)]. Figure 3.6 shows the structure of an RO PUF.

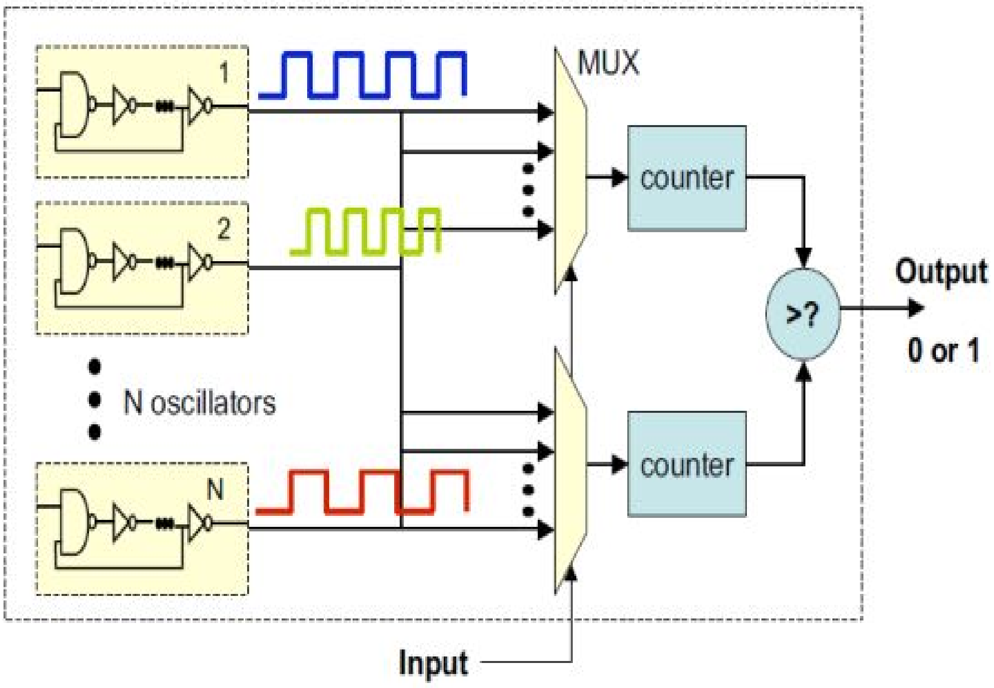


Fig. 3.6: Structure of a RO PUF

To make this weak traditional PUF into a strong one, the comparator is replaced with a logic circuit. By that for n ROs, we can get 2n or 3n bits which results in more CRPs which makes it as a strong PUF.

3.3.3 ARBITER PUF

Arbiter PUF comes under the category of PUFs based on the delay. It requires symmetry in layout. Because the number of CRPs fluctuates with the power of 2 in relation to the number of components, it falls into the strong PUF category. A 128 bit arbiter PUF has 2^{128} CRPs.

Arbiter PUF construction: The circuit of an arbiter PUF have a sequence of 2:1 muxes (which have two inputs and one output) with a D flip flop at the end of it. An n bit arbiter PUF contains 2n number of the muxes. These muxes are arranged such that n muxes are at the top and the remaining muxes are at the bottom. A set of two muxes which are above and below or the two muxes in one column is called as an arbiter switch. The two select lines of the two muxes are combined to form single line where the challenge is given. As a 2:1 mux has inputs i0 and i1 the dissimilar inputs of muxes i.e. say i0 of one mux and i1 of other mux are combined to form a single wire. The circuit of arbiter switch is shown in the Figure 3.7.

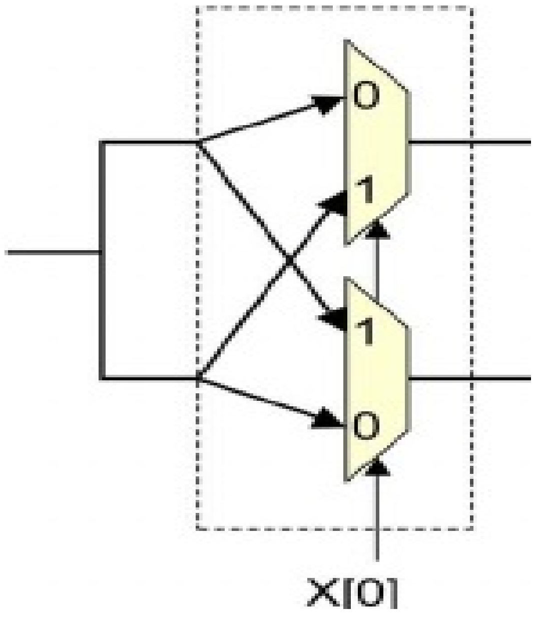


Fig. 3.7: Arbiter Switch

Similarly, a series of n arbiter switches results in a n-bit arbiter PUF. The circuit of arbiter PUF is shown in the Figure 3.8.

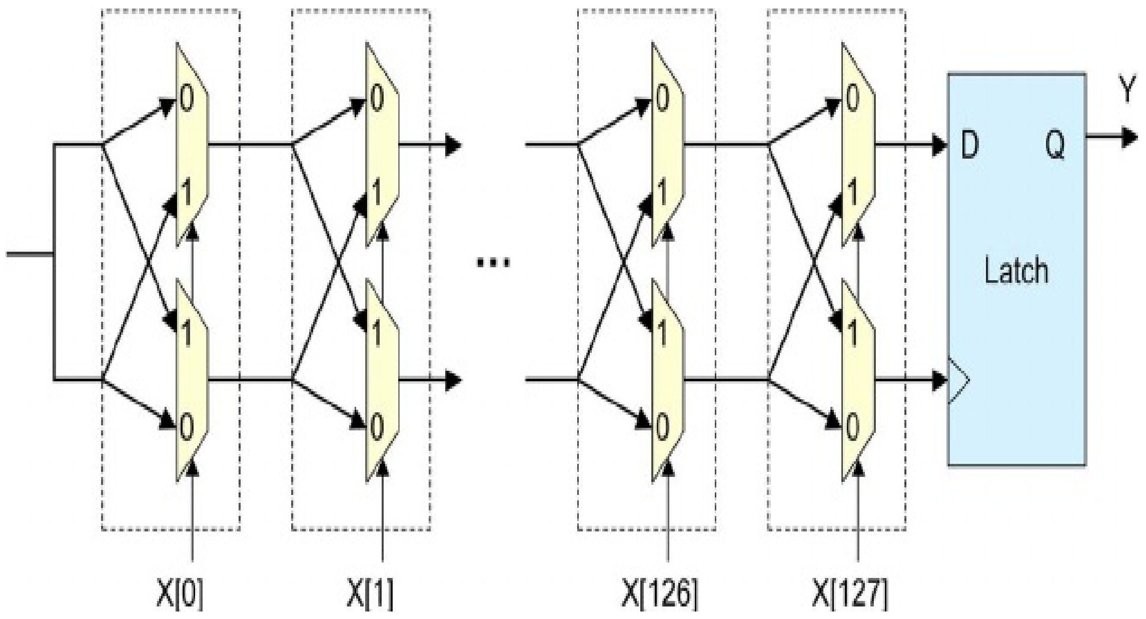


Fig. 3.8: Arbiter PUF

As can be seen in the diagram, at the end of the circuit there is a D flip flop, with top output of final arbiter switch linked to the input of the flip flop and the bottom output of the final arbiter switch linked to the clock pin of the D flip flop.

Working of an arbiter PUF: A rising pulse is given at the input of the arbiter PUF. This pulse travels down the two paths. Each path come across different delays. There is a difference in delay in the two paths because the reason of the delay is the intrinsic imperfections that are present in the muxes. Though all are 2:1 muxes which are similar, the intrinsic process variations formed during the fabrication process makes the delays dissimilar. With the help of challenge bits that are given at the select lines, the paths are swapped. If top pulse arrives first, the output is considered 1, and if the bottom pulse arrives first, the output is considered 0.

This is worked as follows. When the top signal comes first which is connected to D input and the bottom signal comes later which is connected to clock input , then the setup time condition of the D flip flop is satisfied which results giving 1 as output as the input was a rising pulse. Similarly, when the bottom signal comes first, the clock is activated and since there is no input at D flip flop the output we will get is 0. In the Figure 3.8 we can see the circuit of the 128 bit arbiter PUF.

3.3.4 XOR PUF

A xor PUF is generated by taking some number of arbiter PUFs and doing xor operation of all the outputs. This is done to make the circuit more complex and to strengthen the toughness of the architectures of arbiter PUFs. This helps to be more resilient from machine learning attacks. The structure of a xor PUF is shown in the Figure 3.9.

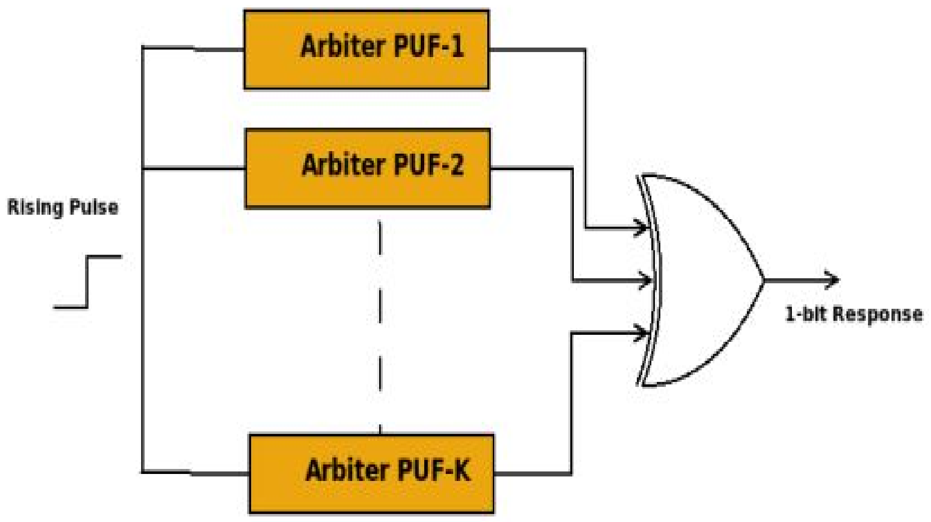


Fig. 3.9: XOR PUF

Working: The working of a xor PUF is similar to that of an arbiter PUF. A rising pulse is given as an input to all the arbiter PUFs at the same time, the signal passes through all the arbiter PUFS encountering different delays that are formed due to inherent process variations resulting a 1 bit response from each arbiter PUF. These responses are applied to a xor gate which gives a 1 bit output which is our response of the xor PUF.

3.4 PERFORMANCE EVALUATION OF PUFS

The performance evaluation of PUFs can be done by using 4 parameters [Maiti *et al.* (2013)].

1 **Uniqueness**

2 **Uniformity**

3 **Reliability**

4 **Bit aliasing**

If the below diagram is taken as the measurement of PUF dimensions as shown in the Figure 3.10, then the four parameters are defined as

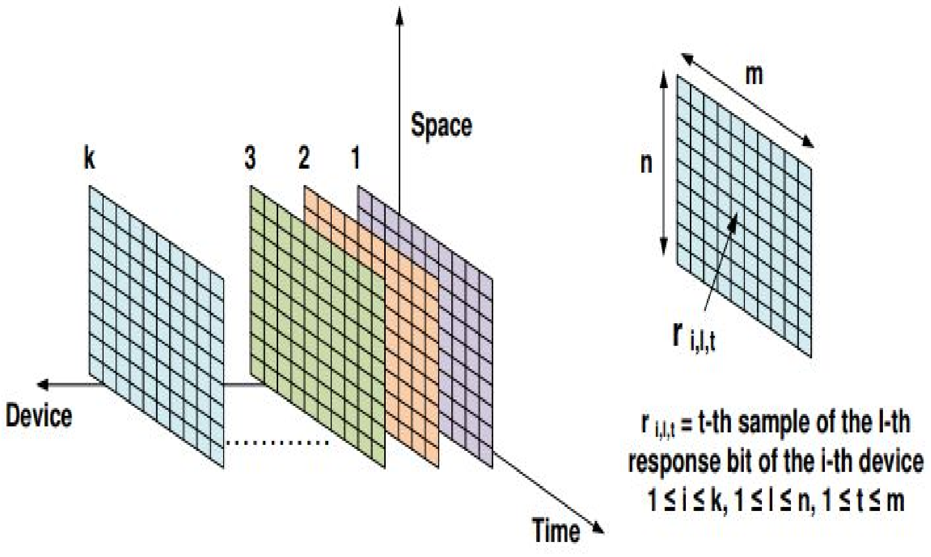

Space
k
3
2
1
Device
Time
m
n
r i,l,t
r i,l,t = t-th sample of the l-th response bit of the i-th device
1 ≤ i ≤ k, 1 ≤ l ≤ n, 1 ≤ t ≤ m

Fig. 3.10: Dimension of a PUF in time and space

3.4.1 UNIQUENESS

PUF's uniqueness is a feature that allows it to provide diverse responses to the same challenge for multiple devices of same type. Calculating the Hamming Distance (HD) between two responses or outputs collected from the two separate devices from the same chip, also known as inter chip hamming distance, is how it's computed. It is written as

$$Uniqueness = \frac{2}{p(p-1)} \sum_{j=1}^{p-1} \sum_{k=j+1}^{p} HD(r_j, r_k)$$

In the equation p represents the total number of devices. For the same challenge, r_j, r_k represents the responses acquired from device j and k . The evaluation of uniqueness is shown in the Figure 3.11.

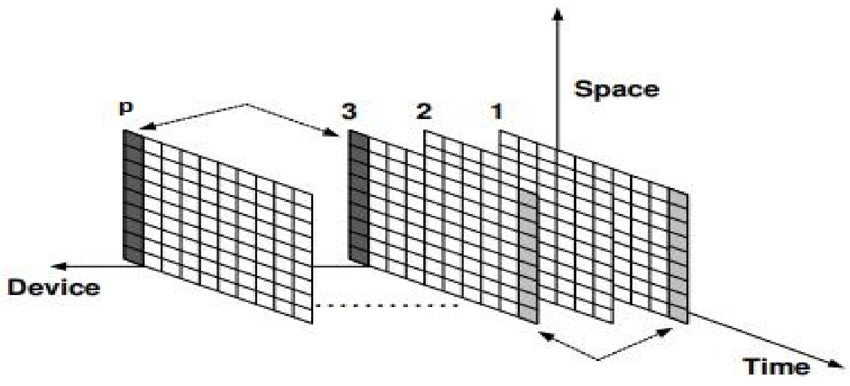


Fig. 3.11: Uniqueness evaluation

3.4.2 UNIFORMITY

It's a metric measuring how many regular 0's and 1's present in a PUF's response. Uniformity in a PUF is commonly assessed in Hamming Weight (HW) of a p-bit response and should be at 50%.Uniformity is described as

$$Uniformity = \sum_{i=1}^{p} r_i$$

In the above equation r_i represents the i^{th} bit of response. The evaluation of uniformity is shown in the Figure 3.12.

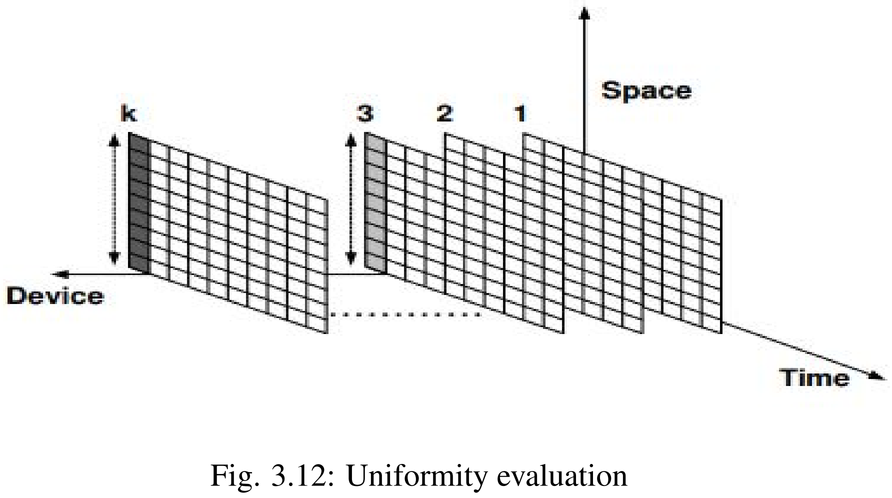


Fig. 3.12: Uniformity evaluation

3.4.3 RELIABILITY

The degree to which PUF delivers a stable output with changes in time, power, and temperature is referred to as reliability. By assessing PUF with the same challenge set, a set of responses is collected, and their intra-chip HD is calculated. It's described as

$$HD_{intra} = \frac{1}{p}\sum_{u=1}^{p} HD(r_j, \hat{r}_j)$$

$$Reliability = 100\% - HD_intra\%$$

In the above equation r_j represents the n-bit response of j^{th} device acquired in normal condition. The evaluation of reliability is shown in the Figure 3.13.

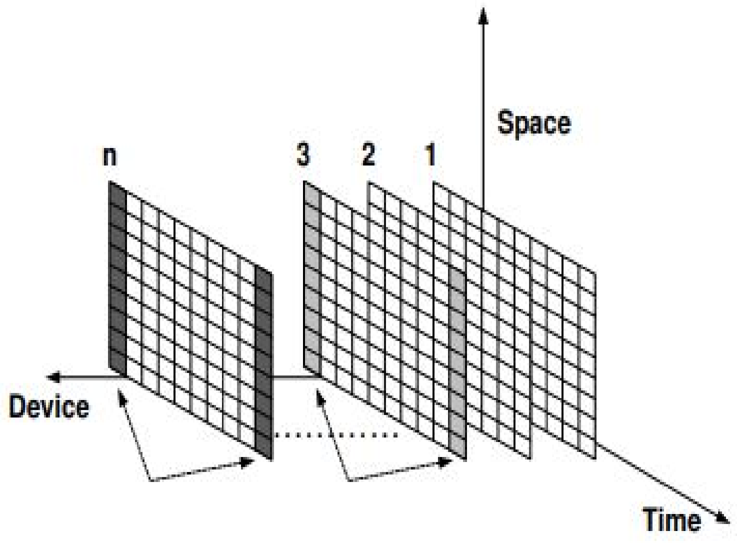


Fig. 3.13: Reliability evaluation

3.4.4 BIT ALIASING

When aliasing of bit happens, we obtain the identical PUF replies from different chips, which is an unwanted outcome. The bit-aliasing caused in the l^{th} bit of a PUF is calculated as the percentage of the Hamming Weight (HW) of the identifying PUF's l^{th} bit across k devices.It is described as

$$(BitAliasing)_l = \frac{1}{k}\sum_{j=1}^{k} r_{j,l} \times 100\%$$

In the above equation from a chip i , $r_{j,l}$ represents the l^{th} binary bit of n bit response. The evaluation of bit aliasing is shown in the Figure 3.14.

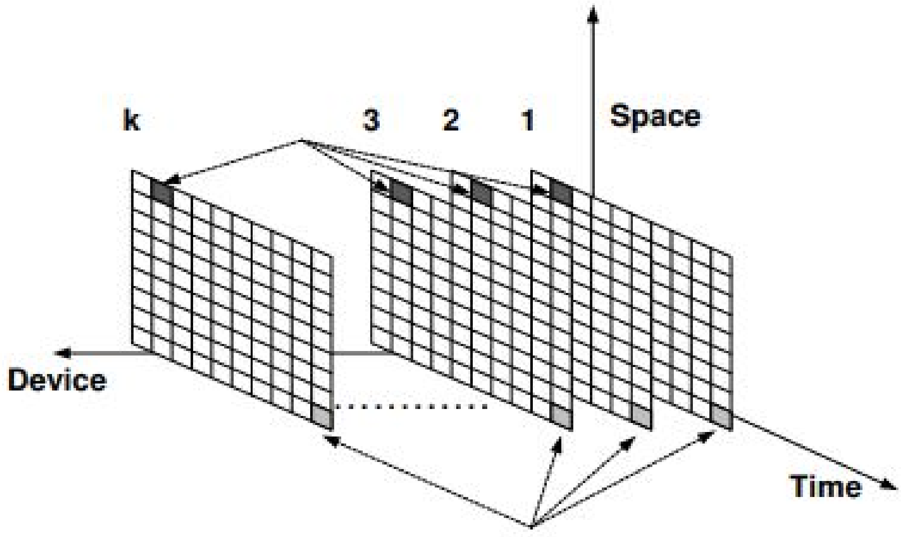


Fig. 3.14: Bit Aliasing evaluation

CHAPTER 4

PUF BASED AUTHENTICATION

4.1 TRADITIONAL PUF BASED AUTHENTICATION

Instead of using various cryptographic algorithms we can use PUFs for the authentication of devices and also with the use of PUFs this can be made possible with the less hardware. With the help of CRPs that are generated in the PUF a device establishes its unique identity from other devices and makes authentication possible. The device which needs the information is called prover because it needs to prove itself to get information. The device which provides the information is called the verifier as it verifies the data provided by the prover.
A device can become prover as well as a verifier according to the need.
As can be observed there is a scope for the attacker to intercept the authentication process [Barbareschi *et al.* (2018)].

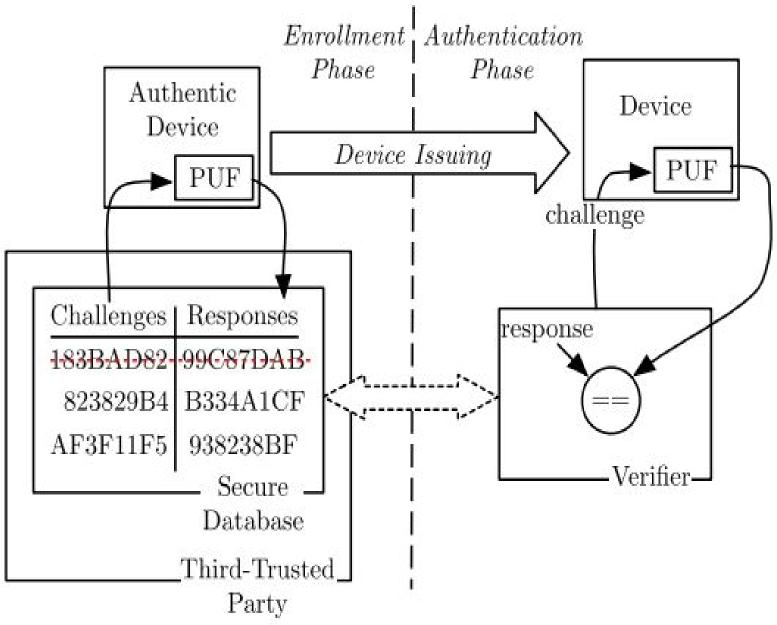


Fig. 4.1: PUF based authentication mechanism

The authentication process composed of two steps:

1 **Enrolment:**
This is the first step in the PUF based authentication. Here the devices get enrolled with CRPs. In this step, for a group of random challenges the responses are generated from the PUF of verifier and are stored in the memory of verifier.

2 **Verification:**
This is performed after enrolment step. Whenever prover invokes verifier for authentication, a challenge which is not previously used and the corresponding response is taken and the challenge is given as input to the PUF of prover device. When the response is generated, it is compared with the response extracted from the verifier's memory. If the both are equal then the authentication is successful and the transfer of information happens. .

The important thing that needs to be noted is that the challenge once used for the authentication should not be used another time to be secured from the attackers. The PUF based authentication mechanism can be seen in the Figure 4.1.

4.2 IMPROVED PUF BASED AUTHENTICATION

The improved PUF based authentication solves the issue of using a challenge only once to avoid malicious attacks from the attacker. Since the problem is arising at the challenge, instead of pre generating challenge and storing it can be done in such a way that the challenge is generated randomly on the spot. The challenge can be generated randomly through the use of Linear Feedback Shift Register (LFSR).

In this authentication also it consists of 2 phases:

1 **Enrolment:**
This is the first step in the improved PUF based authentication. Here the devices get enrolled with CRPs. If the PUFs in both the devices are n bit then a random n/2 bit number is generated from the prover and also the verifier. These both n/2 bit numbers concatenates forming n bit number. This is mapped to an index by using mod operator in doing (n bit generated number) mod n. Based on the index, LFSR is initiated with a seed value and it runs that many number of times as the index says so, to generate challenges. For the respective index the generated response from the PUF of verifier are stored in the memory of verifier. It is shown in the Figure 4.2.

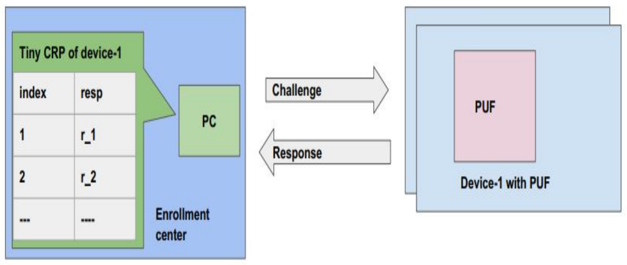


Fig. 4.2: Enrolment phase

2 **Verification:**
This is performed after enrolment step. Whenever prover invokes verifier for authentication, an index and the corresponding response is taken and the index is given as input to the LFSR. LFSR runs the number of times that index said and generates a challenge which gives as an input to the PUF of prover device. When the response from prover device is generated, it is compared with the response extracted from the verifier's memory. If the both are equal then the authentication is successful and the transfer of information happens. It is shown in the Figure 4.3.

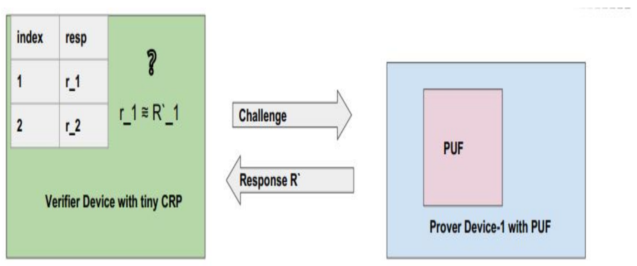


Fig. 4.3: Verification phase

Thus the authentication happens without the need of storing challenge as challenge can be generated instantly. Authentication security is directly proportional to the amount of CRPs present. The bigger the number of CRPs, the more secure the system. A strong PUF can allow for a large number of CRPs.

So, in this project the work is done on the **128 bit Arbiter PUF**.

CHAPTER 5

INTELLECTUAL PROPERTY AND MODULE USED IN THE PROJECT

5.1 VIO

VIO expanded form is Virtual Input Output. It is an inbuilt customizable IP that is present in the Vivado itself. In the real time VIO is used to give inputs and observe outputs. According to the design and FPGA, the width of inputs and outputs can be customized. The difference between ILA and VIO is that in VIO there is no need of n chip or off chip ram.

There are two types of probes in VIO. They are

1 **Input probes:**
They act as input to the VIO. While doing connections the output of the circuit or our design is connected to the input of VIO.

2 **Output probe:**
They act as output to the VIO. While doing connections the inputs of the circuit is connected to the output of the VIO.

The block diagram of VIO is shown in the Figure 5.1.

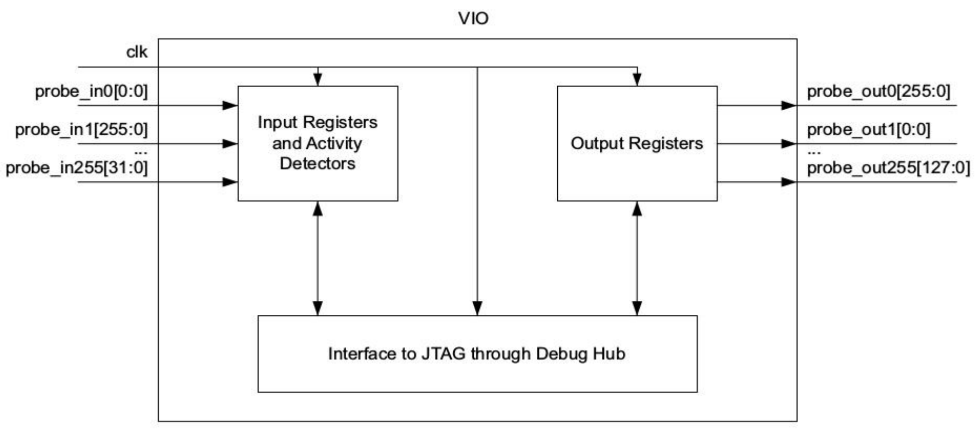
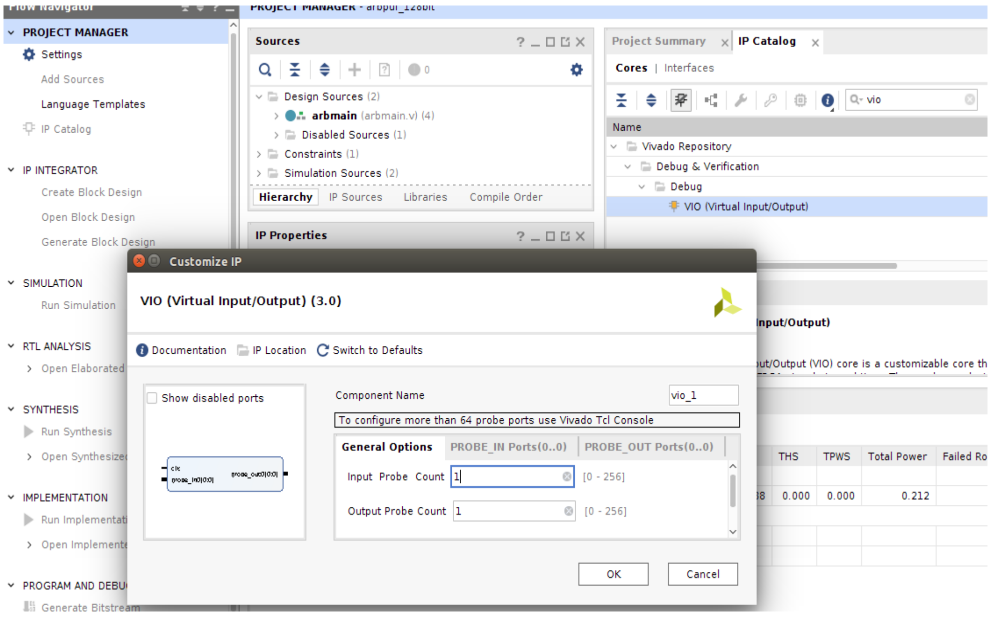


Fig. 5.1: Block diagram of VIO

In the Vivado the VIO IP can be generated as follows: At the project manager which is in the left side pane, after clicking on the IP catalog a window is opened where if u search vio and click on it. VIO is opened. There we can the width of input and output probes and generate the VIO.

The VIO IP is shown in the Figure 5.2.


Customize IP

VIO (Virtual Input/Output) (3.0)

Component Name: vio_1

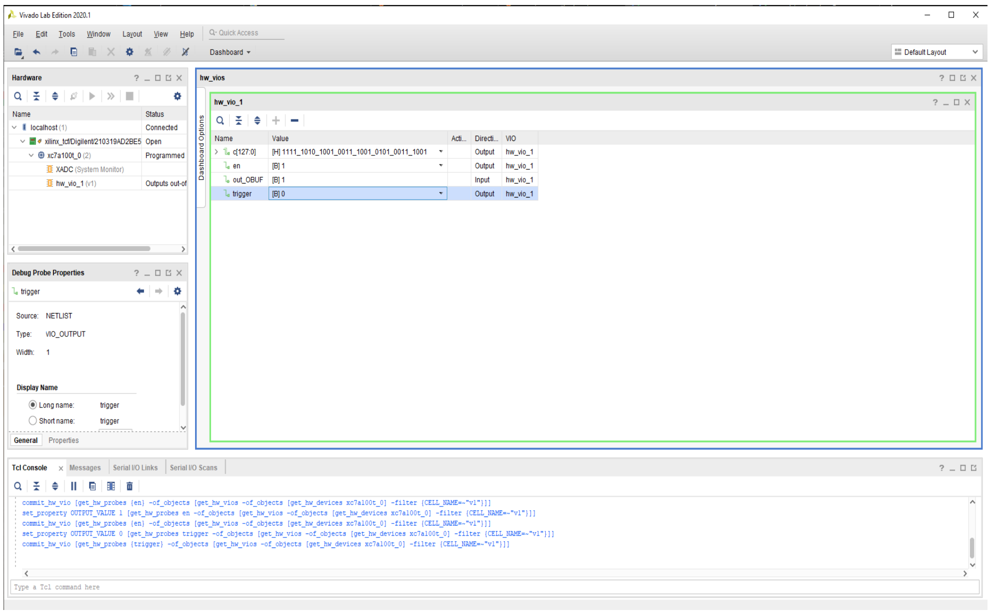
To configure more than 64 probe ports use Vivado Tcl Console

General Options | PROBE_IN Ports(0..0) | PROBE_OUT Ports(0..0)

Input Probe Count	1	[0 - 256]
Output Probe Count	1	[0 - 256]

Fig. 5.2: VIO IP

The GUI of VIO after opening the hardware manager is shown in the Figure 5.3.


hw_vio_1

Name	Value	Acti...	Directi...	VIO
c[127:0]	[H] 1111_1010_1001_0011_1001_0101_0011_1001		Output	hw_vio_1
en	[B] 1		Output	hw_vio_1
out_OBUF	[B] 1		Input	hw_vio_1
trigger	[B] 0		Output	hw_vio_1

Debug Probe Properties: trigger

Source: NETLIST
Type: VIO_OUTPUT
Width: 1

Tcl Console

```
commit_hw_vio [get_hw_probes {en} -of_objects [get_hw_vios -of_objects [get_hw_devices xc7a100t_0] -filter {CELL_NAME=~"v1"}]]
set_property OUTPUT_VALUE 1 [get_hw_probes en -of_objects [get_hw_vios -of_objects [get_hw_devices xc7a100t_0] -filter {CELL_NAME=~"v1"}]]
commit_hw_vio [get_hw_probes {en} -of_objects [get_hw_vios -of_objects [get_hw_devices xc7a100t_0] -filter {CELL_NAME=~"v1"}]]
set_property OUTPUT_VALUE 0 [get_hw_probes trigger -of_objects [get_hw_vios -of_objects [get_hw_devices xc7a100t_0] -filter {CELL_NAME=~"v1"}]]
commit_hw_vio [get_hw_probes {trigger} -of_objects [get_hw_vios -of_objects [get_hw_devices xc7a100t_0] -filter {CELL_NAME=~"v1"}]]
```

Fig. 5.3: VIO GUI

5.2 CLOCKING WIZARD

It is an inbuilt customizable IP that is present in the vivado itself. Clocking wizard IP is a crucial IP which drives our design, LFSR, and VIO IP. This IP is used to generate clock which satisfies the design requirements. We can set the sampling frequency according to our requirement.

The block diagram of clocking wizard is shown in the Figure 5.4.

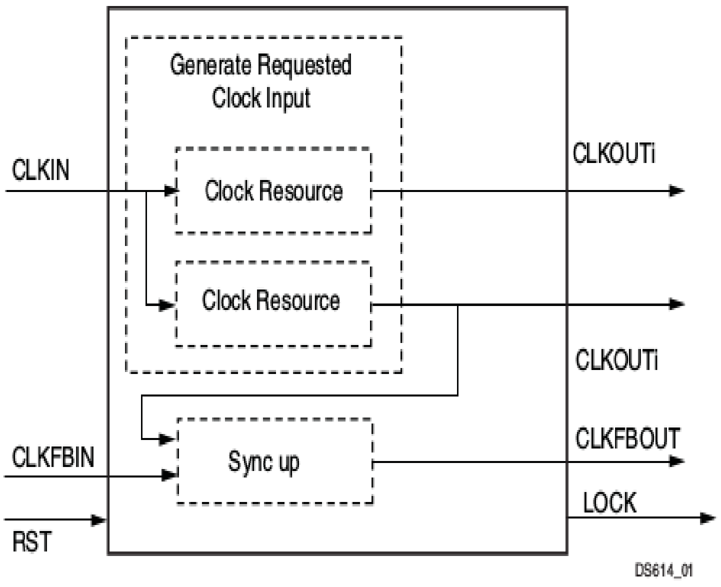
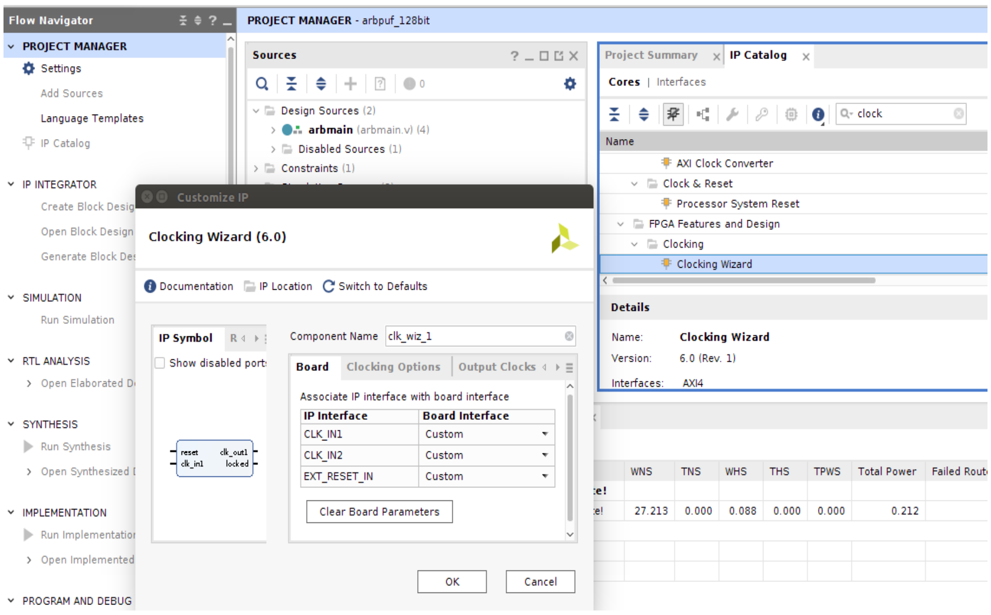


Fig. 5.4: Block diagram of clocking wizard

<u>In the Vivado the clocking wizard IP can be generated as follows:</u>
At the project manager which is in the left side pane, after clicking on the IP catalog a window is opened where if u search clocking wizard and click on it. Clocking wizard is opened. Over there we can set reset signal, locked signal etc and the sampling frequency as per our requirement.

The clocking wizard IP is shown in the Figure 5.5.



Component Name clk_wiz_1

Associate IP interface with board interface

IP Interface	Board Interface
CLK_IN1	Custom
CLK_IN2	Custom
EXT_RESET_IN	Custom

Details

Name: Clocking Wizard
Version: 6.0 (Rev. 1)
Interfaces: AXI4

WNS	TNS	WHS	THS	TPWS	Total Power	Failed Rout
27.213	0.000	0.088	0.000	0.000	0.212	

Fig. 5.5: Clocking wizard IP

5.3 LFSR

LFSR expanded form is Linear Feedback Shift Register. A LFSR consists of series of shift registers connected with each other. Output of the last register with outputs of some other register is given to a logic circuit where the output of the logic circuit is given as the input of the first register. The logic of the logic circuit is defined by the primitive polynomial. LFSR should have an initial value as all the shift registers in LFSR should have an initial state which is called seed value. Since from the circuit we can observe that there is linear feedback from the connection of output to input.

A n bit LFSR contains n shift registers and it should be initialised with n bit seed value. With the respective shifts for each clock cycle the maximum number of runs that an LFSR can run is $2^n - 1$ times. Thus the periodicity of LFSR becomes $2^n - 1$.

In the project for generating challenges for 128 bit Arbiter PUF, a 128 bit LFSR is used. For the 128 bit LFSR the primitive polynomial $X^{128} + X^{127} + X^{126} + X^{121} + 1$ is used. Particularly this primitive polynomial is used to generate LFSR as it generates the most random outputs.

The figure of 128 bit LFSR with the primitive polynomial applied is shown in the Figure 5.6.

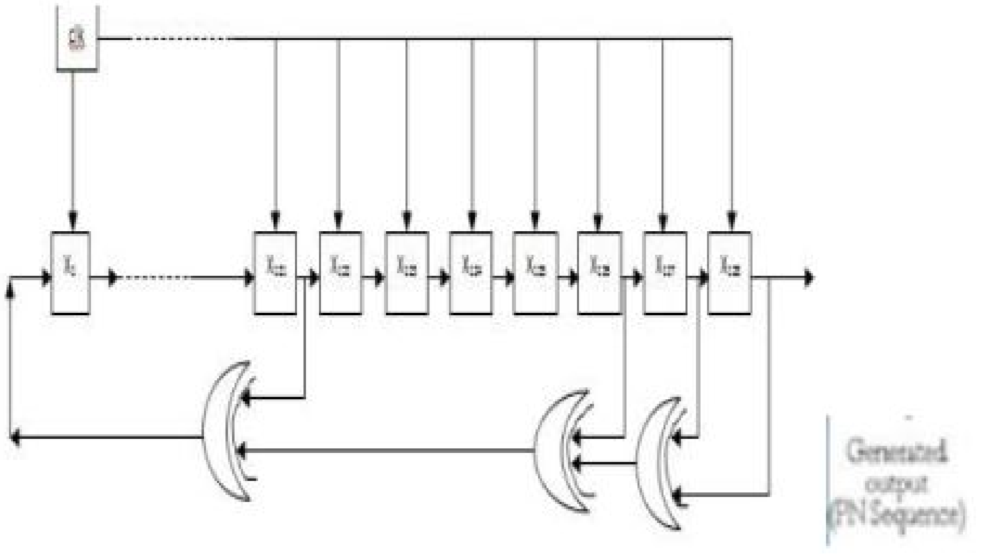


Fig. 5.6: 128 bit LFSR with primitive polynomial $X^{128} + X^{127} + X^{126} + X^{121} + 1$

CHAPTER 6

BIRD VIEW OF THINGS DONE IN THE PROJECT

In this project a 128 bit arbiter PUF is implemented on the ARTY A7-100 FPGA.

First the Verilog code of 8 bit arbiter PUF was written. Clocking wizard and VIO IPs are generated. Since an arbiter PUF requires layout symmetry, manual placement of LUTs in a FPGA is done. After that the synthesis, implementation were done. After knowing that the bit stream is generated successfully. I have written a python code to generate a parameterized arbiter PUF and automatically a vivado project is generated.

What happens when we run the python code:
When we run the python code it first asks for the value of N to generate N bit arbiter PUF. When a value of N is given it asks to input the name of the vivado project that needs to be created. After given the name of the project it asks for the location where all the generated files need to be stored.

After giving the location, it then runs and generates all Verilog files and a xdc file in which manual placement of modules in FPGA information is present of the arbiter PUF. Then automatically it takes all the Verilog files and the xdc file generates the project. It also automatically generates the clocking wizard IP and the VIO IP and get synthesized. That's what happens when we run the python code.

Project work done:
In this project a 128 bit arbiter PUF is implemented on the ARTY A7-100 FPGA.
For this we run the python code. When it asks for the value of N, we give input as 128. Then we enter the name of the project and the location where all the files need to be stored. After we open the project we can see all the verilog files and xdc fie for 128 bit arbiter PUF have already been added and the IPs of clocking wizard and VIO

have already been generated and synthesized. In this project we have two ways to give challenges to arbiter PUF.

1 From LFSR that is generated from verilog code. A seed value should be given as input and the primitive polynomial must be mentioned. The circuit diagram of the whole project when LFSR used is shown in the Figure 6.1.

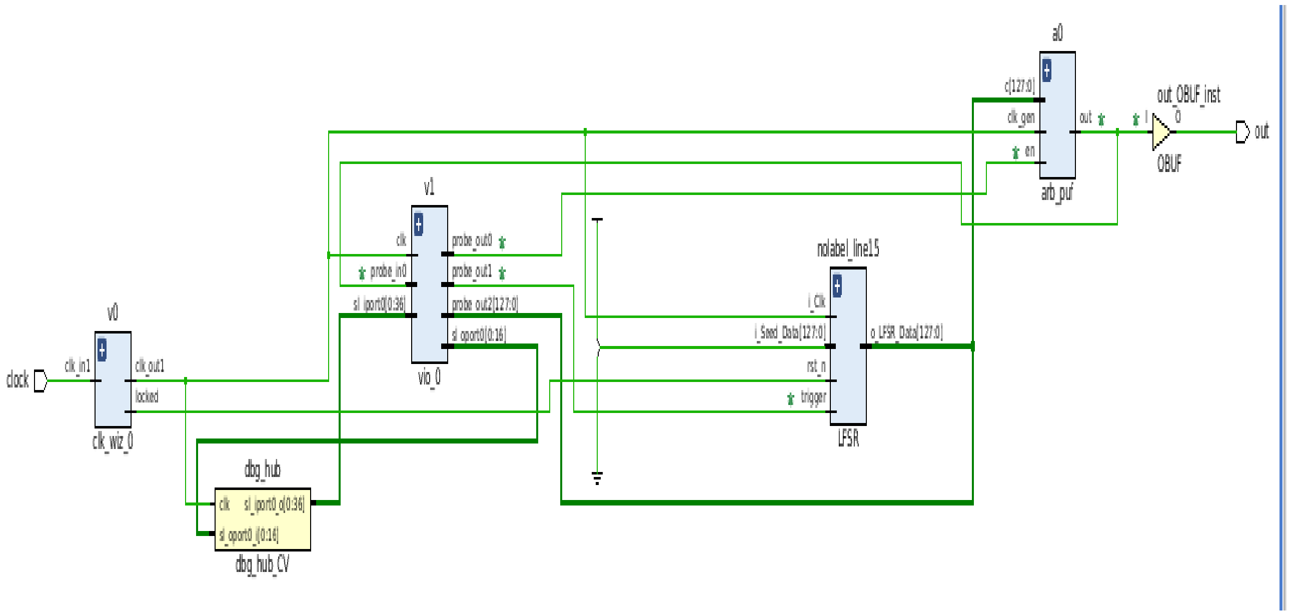


Fig. 6.1: Circuit diagram of the project

2 From the random number generator which was coded using tcl language. A seed value should be given and also should mention the number of times the random values are needed.

Since in this project 2 is a viable option because with this we can both collect the random challenges and the outputs produced.

So in this project we use 2.

After opening project we can directly generate bit stream. After that we connect ARTY A7–100 FPGA and open the hardware manager. We run the tcl code present in the document and text files is created where it has the generated random challenges which are given to the input of 128 bit arbiter PUF and also the generated responses.

As mentioned about manual placement of modules in FPGA, a detailed procedure of how to do manual placement of modules using FPGA was mentioned in the chatper 7. The codes that i have written for the project are present in this document.

CHAPTER 7

MANUAL PLACEMENT OF MODULES ON FPGA USING PBLOCKS

Our main aim here is that we need to manually place the modules or components of our verilog code so that we can accurately place them wherever needed.

For doing this we need to use something known as pblock.

What is a Pblock?

Pblocks are rectangles that are used to contain logic for floor plan. They are used during floor planning placement to group related logic and assign it to a region of the target device.

How do we create a pblock?

We can draw or create a pblock either by coding in xdc file or by the use GUI (graphical user interface) that is present in Vivado. If we draw a pblock by GUI and save it, automatically a .xdc file along with code is created. So, if we want any changes we can either do by changing code in .xdc file or by changing in GUI. So, finally the easiest way to create a pblock is by GUI. Before getting to know more about pblock it is needed to know about configurable logic block of a FPGA as this is needed down the line. Here the example of FPGA taken is 7 series FPGA.

Brief Overview of CLB of a 7 series FPGA:

A configurable logic block (CLB) consists of two slices(slice_M, slice_L) which are present side by side. slice_M is memory capable and slice_L is capable of logic and carry.

There are four 6-input LUTs present per slice. A Single LUT in slice_M can be a 32 bit shift register or 64X1 RAM.

There are 2 flip flops per LUT.

The figure of a slice can be shown in the Figure 7.1.

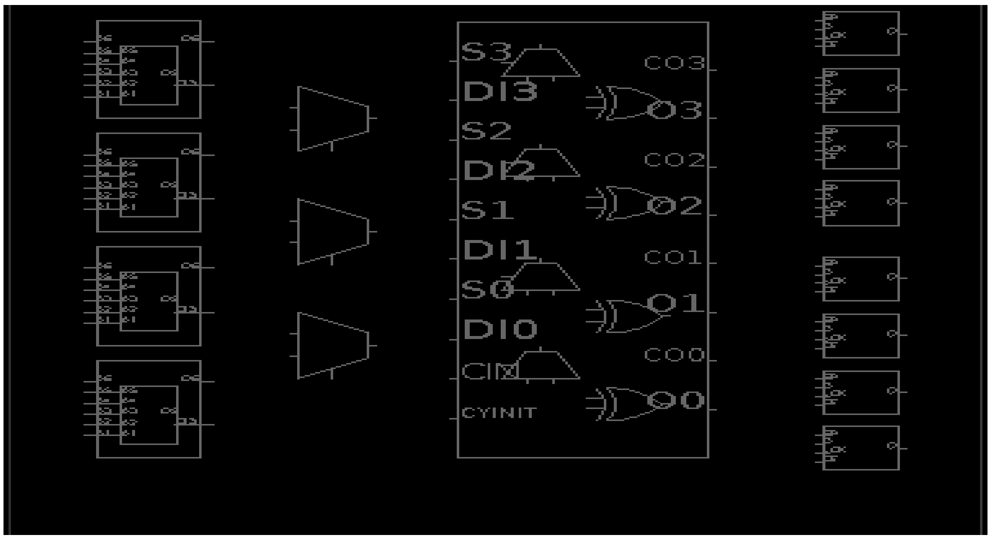
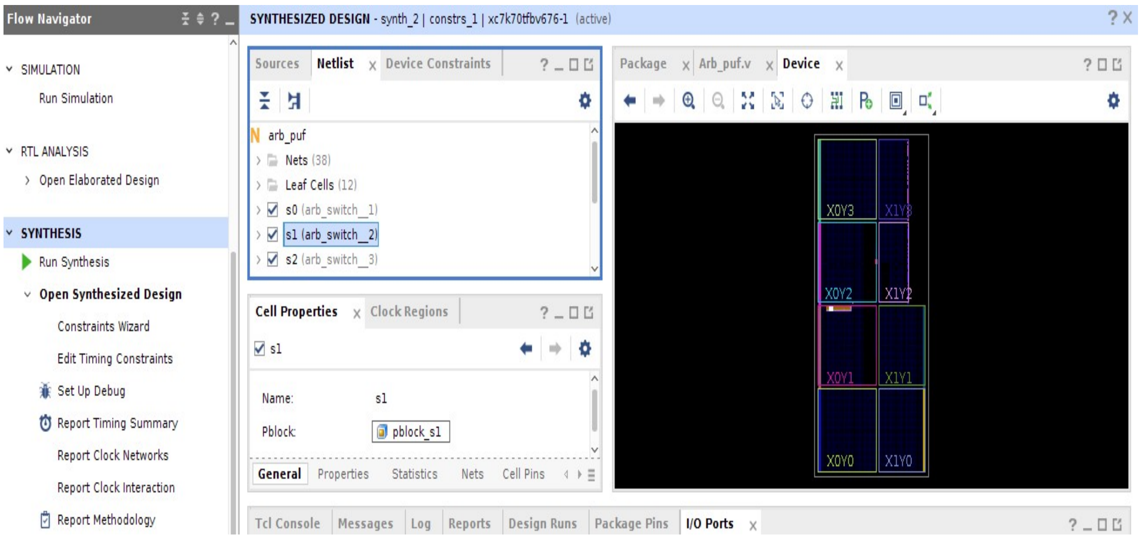


Fig. 7.1: A Slice

How to create a pblock in GUI?

Before that we need to synthesize the verilog code that is written and we need to click open synthesized design in the left pane. In the middle pane where we have tabs such as sources, netlist. We need to click on netlist tab and there we can see the module names that are formed from our design. In my example, we can see s0, s1..s7. So our aim is to do placement of these modules manually. We need to create a pblock for each module and assign them to their respective pblocks. The figure of GUI where we can create a pblock is shown in the Figure 7.2.

In the right pane select the device tab. To create a pblock right click on the module for which we want to create, select floor planning and select draw pblock. After clicking that take the cursor to device tab (which is in right pane) and where ever needed by clicking left mouse click and holding ,drag the mouse to determine the size needed and release. A window to name the pblock is flashed. Set a name. Thus a pblock is created.



SYNTHESIZED DESIGN - synth_2 | constrs_1 | xc7k70tfbv676-1 (active)

arb_puf
Nets (38)
Leaf Cells (12)
s0 (arb_switch_1)
s1 (arb_switch_2)
s2 (arb_switch_3)

s1

Name: s1
Pblock: pblock_s1

Fig. 7.2: GUI where pblocks can be created

After pblock is created, right click the same module name that was done earlier, select floor planning and select assign to pblock. A window with different pblocks that were created is opened. Select the same pblock that was created for this module and click ok. Thus the respective pblock is assigned to that respective module.

A pblock created is shown in the Figure 7.3.

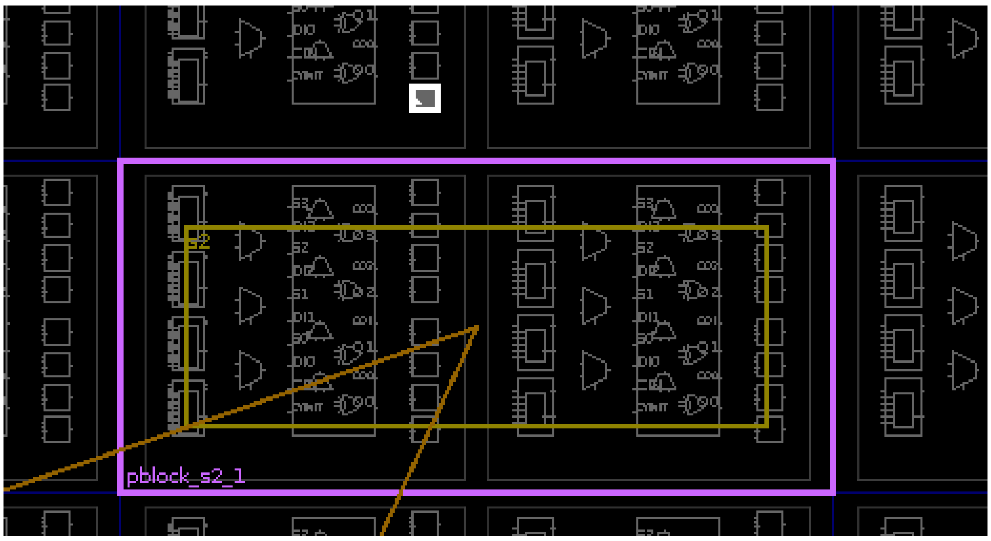


Fig. 7.3: A pblock

The violet rectangle is the pblock.

That is how we do the same process for all the modules.

It is shown in the Figure 7.4.

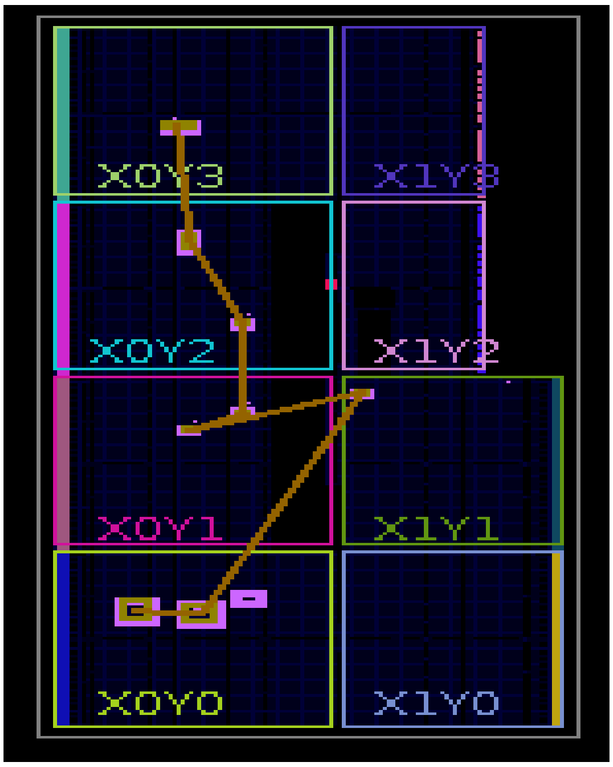


Fig. 7.4: Pblocks of different modules

After all the pblocks are created click the save button. When done that a window to set name for constraints file (xdc file) is popped up and set the name of that xdc file. After doing that xdc file is created and if we open the xdc file we can see the code of what all that was done. To do changes of pblock we can either do by GUI (as in above figure) or by doing changes in code.

After doing that a doubt arises that what is the smallest size of pblock?

what is the smallest size of pblock?

The smallest size of a pblock is size of one CLB i.e 2 slices(in case of 7 series FPGA). Beyond that we cannot reduce the size of pblock. It is shown in the Figure 7.5.

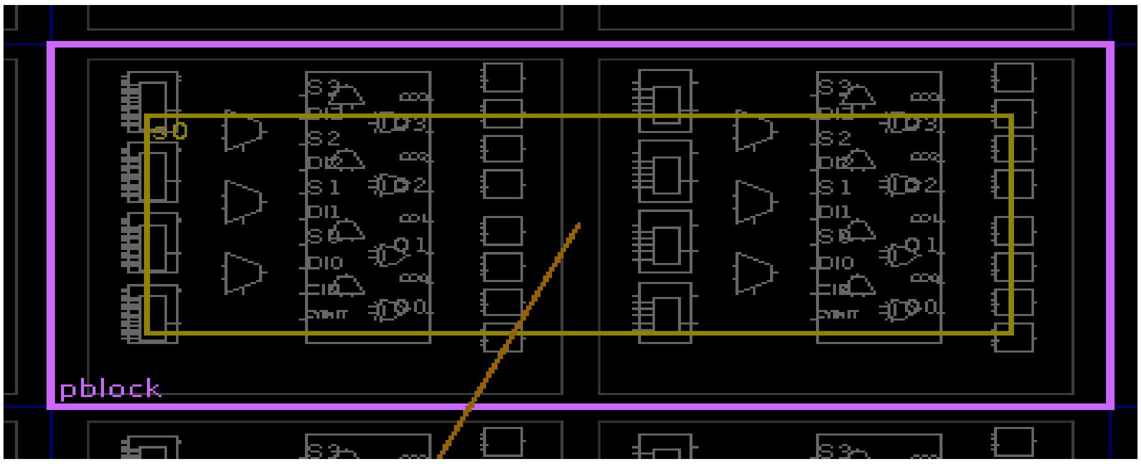
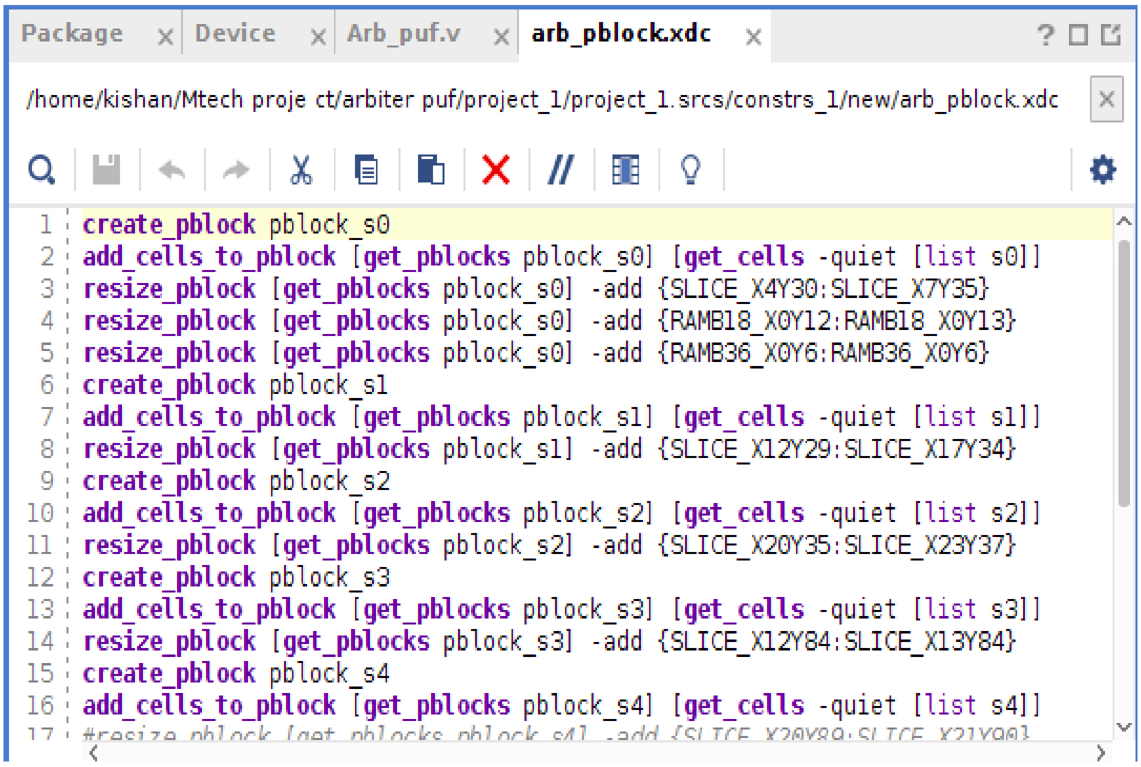


Fig. 7.5: Smallest pblock



/home/kishan/Mtech proje ct/arbiter puf/project_1/project_1.srcs/constrs_1/new/arb_pblock.xdc

```
1  create_pblock pblock_s0
2  add_cells_to_pblock [get_pblocks pblock_s0] [get_cells -quiet [list s0]]
3  resize_pblock [get_pblocks pblock_s0] -add {SLICE_X4Y30:SLICE_X7Y35}
4  resize_pblock [get_pblocks pblock_s0] -add {RAMB18_X0Y12:RAMB18_X0Y13}
5  resize_pblock [get_pblocks pblock_s0] -add {RAMB36_X0Y6:RAMB36_X0Y6}
6  create_pblock pblock_s1
7  add_cells_to_pblock [get_pblocks pblock_s1] [get_cells -quiet [list s1]]
8  resize_pblock [get_pblocks pblock_s1] -add {SLICE_X12Y29:SLICE_X17Y34}
9  create_pblock pblock_s2
10 add_cells_to_pblock [get_pblocks pblock_s2] [get_cells -quiet [list s2]]
11 resize_pblock [get_pblocks pblock_s2] -add {SLICE_X20Y35:SLICE_X23Y37}
12 create_pblock pblock_s3
13 add_cells_to_pblock [get_pblocks pblock_s3] [get_cells -quiet [list s3]]
14 resize_pblock [get_pblocks pblock_s3] -add {SLICE_X12Y84:SLICE_X13Y84}
15 create_pblock pblock_s4
16 add_cells_to_pblock [get_pblocks pblock_s4] [get_cells -quiet [list s4]]
17 #resize_pblock [get_pblocks pblock_s4] -add {SLICE_X20Y89:SLICE_X21Y90}
```

Fig. 7.6: A view inside xdc file

In the above Figure 7.6 we can see the code in a xdc file. If observed line 6 is for creating pblock. We can change the size of pblock by changing the slices x, y coordinates. For example lets take for pblock_s1. As can be seen we can resize by changing in line 8. In line 8 sliceX12Y29:sliceX17Y34 represents that pblock consists of total 36 slices is shown in the Figure 7.7.

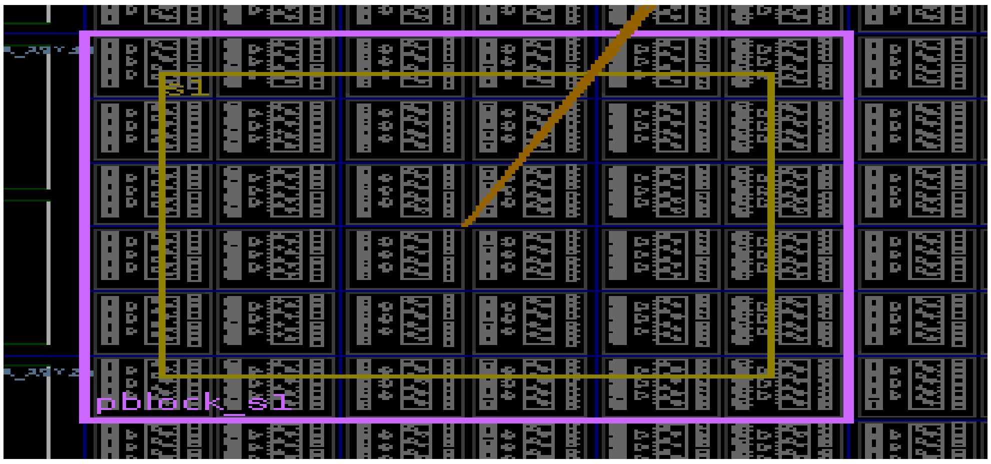


Fig. 7.7: pblock_s1 containing 36 slices

How to know the names of LUTs(Look UP Tables) on which modules that are designed are placed?

To know this we should click Run Implementation that can be seen on left pane. Click open implemented design. In the middle pane open net list tab and in the right pane open device tab. There we can see how the connections are made in the Figure 7.8.

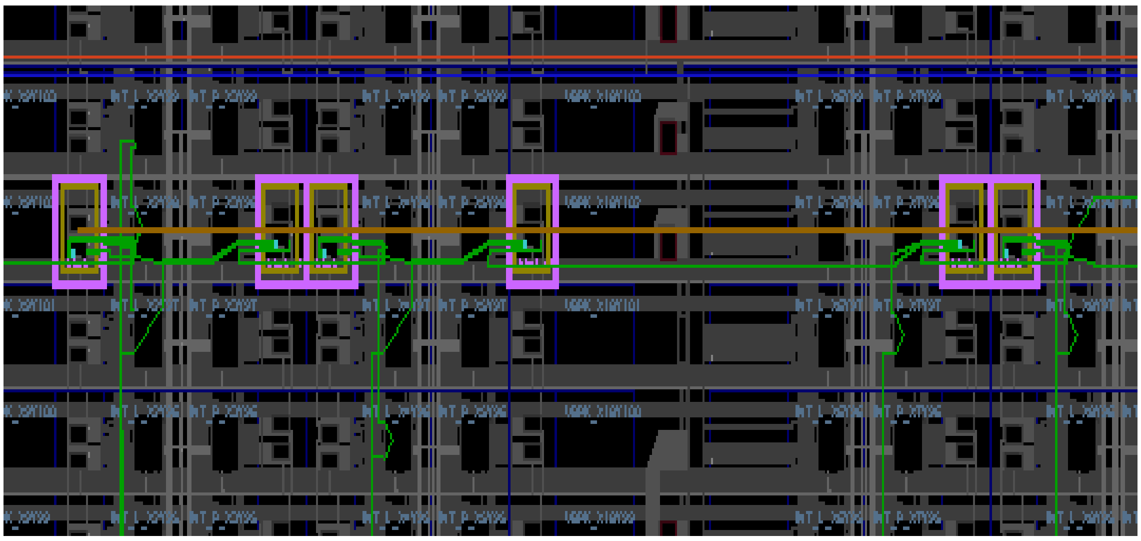


Fig. 7.8: Wire connections between modules

How to see the signals, inputs, outputs of our design?

The wires of the selected signals are highlighted which is shown in the Figure 7.9.

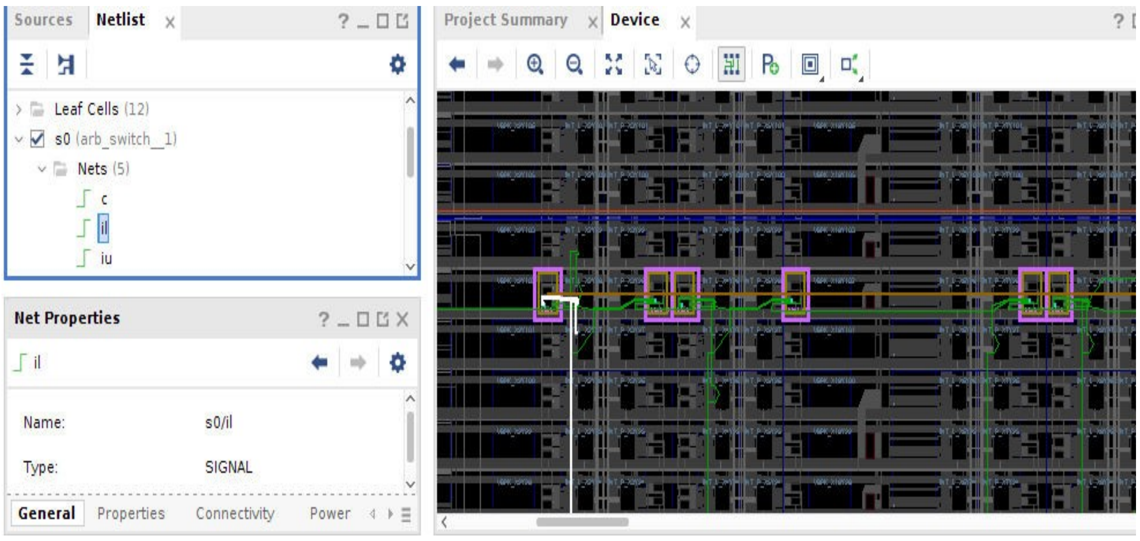
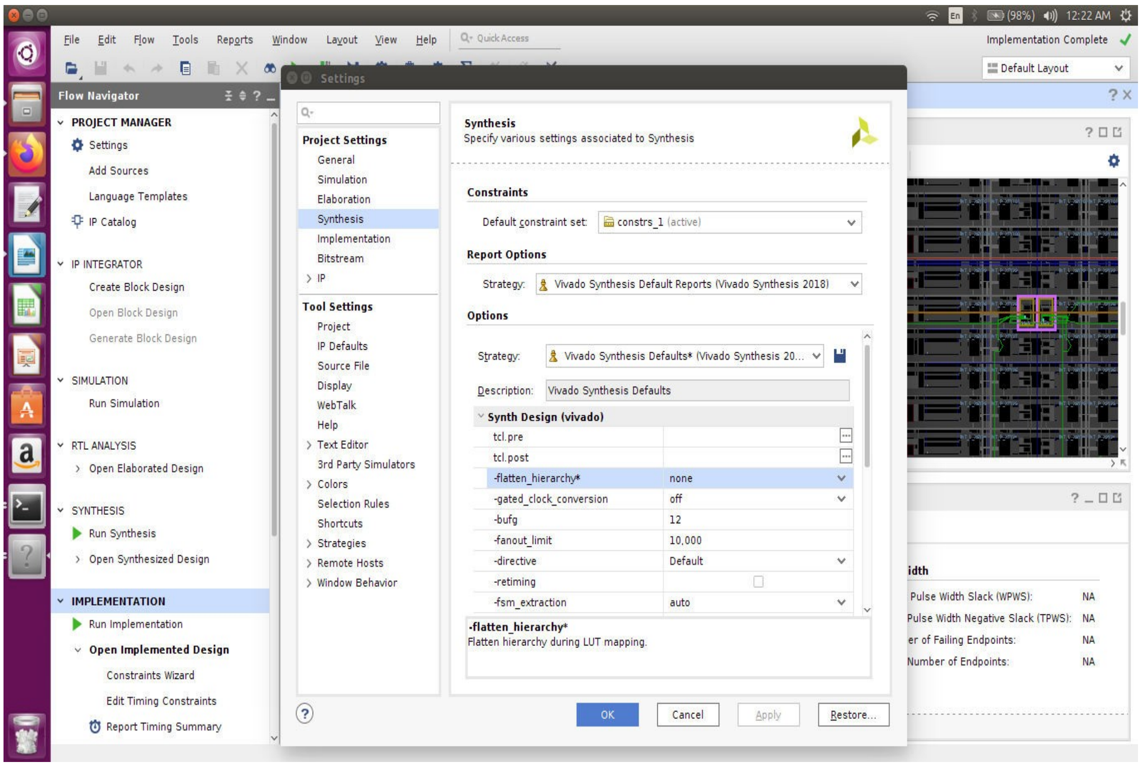


Fig. 7.9: Highlighted wires of selected signals

Things to be noted while writing verilog code for a better manual placement:
To do this manual placement or whatever related to synthesis, the hardware of the code we write should be realised. For that we need to write verilog code mostly in structural representation.

Sometimes, even after writing in structural form and if we run synthesis, sometimes vivado optimizes the hardware and some of the hardware or modules that need to be there might not be present. Then we need to do these following things.

1 Click on the settings that is present in the left pane. Then click on synthesis and set the flatten_hierarchy to none. And click OK. It is shown in the Figure 7.10.



Synthesis
Specify various settings associated to Synthesis

Constraints
Default constraint set: constrs_1 (active)

Report Options
Strategy: Vivado Synthesis Default Reports (Vivado Synthesis 2018)

Options
Strategy: Vivado Synthesis Defaults* (Vivado Synthesis 20...
Description: Vivado Synthesis Defaults

Synth Design (vivado)	
tcl.pre	
tcl.post	
-flatten_hierarchy*	none
-gated_clock_conversion	off
-bufg	12
-fanout_limit	10,000
-directive	Default
-retiming	☐
-fsm_extraction	auto

-flatten_hierarchy*
Flatten hierarchy during LUT mapping.

idth	
Pulse Width Slack (WPWS):	NA
Pulse Width Negative Slack (TPWS):	NA
er of Failing Endpoints:	NA
Number of Endpoints:	NA

Fig. 7.10: Setting flatten_hierarchy to none

2 Open the verilog code that was written. We need to set an attribute. Type (* KEEP = "TRUE" *) at the starting of the line of which that should be realised in the hardware or for the module of which the hardware is getting optimised.

By doing the above things we can make the vivado tool not optimising our design thereby not distorting the intended design.

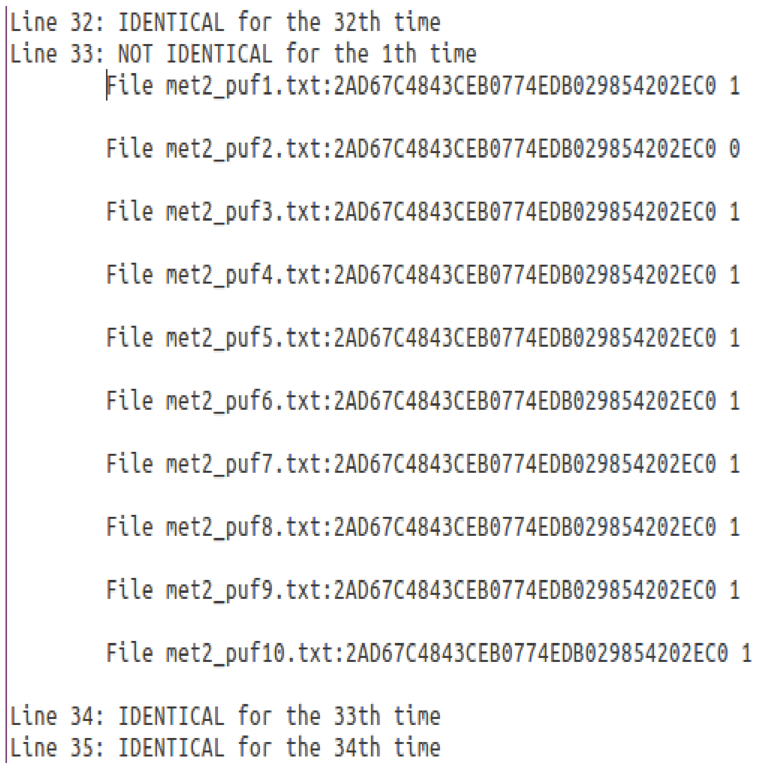
That is how pblocks are created and by being with some caution, we can realise the hardware exactly how we intended and can do manual placement of that as needed.

CHAPTER 8

RESULTS

PUF is a device used for authentication. When the same inputs were given for the PUF of the same FPGA , ideally it should provide the same outputs each and everytime. When the same inputs were applied for the different PUFS present on FPGAs, ideally it should provide the different outputs each and everytime.
ARTY A7-100 FPGA board is used for the implementation of 128 bit Arbiter PUF. For performing the experiment five boards of the ARTY A7-100 family with serial nos: DAD2BB7, DAD2AFA, DAD2CBB, DAD2CF1 and DAD2BDB were taken.
For each board of the ARTY A7-100, 1000 same random challenges were applied for 10 times and the generated responses along with the applied challenges were stored in the text files. This results in the generation of 10 text files with 1000 CRPs in each. Observations were done by comparing the text files.

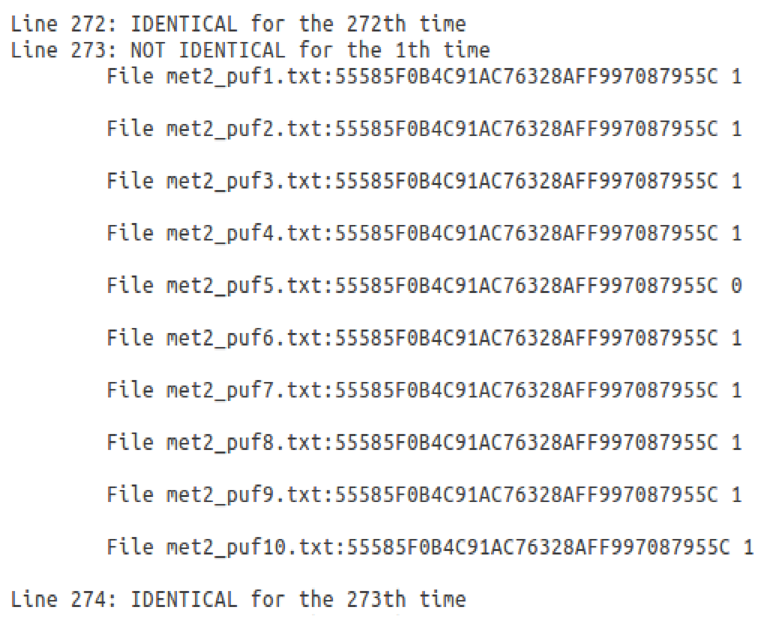
Expected result: When the same inputs were given for the PUF of the same FPGA , ideally it should provide the same outputs each and everytime.
The observations for the boards are as follows:
DAD2BB7: For the 1000 CRPs that are present in each file, 994 CRPs of all the 10 files are identical. When the remaining 6 non identical CRPs of 10 files are observed, it is in such a way that for a non identical CRP, only one response bit from one file among 10 response bits of 10 files is different. Remaining 9 response bits are same. It is shown in the Figure 8.1. Just like the Figure 8.1, in all the 6 non identical CRPs only 1 response bit from one file is different. From this it can be said that the implemented PUF is very much accurate in producing results near to ideal expected results. The results can be seen in this document.



```
Line 32: IDENTICAL for the 32th time
Line 33: NOT IDENTICAL for the 1th time
        File met2_puf1.txt:2AD67C4843CEB0774EDB029854202EC0 1

        File met2_puf2.txt:2AD67C4843CEB0774EDB029854202EC0 0

        File met2_puf3.txt:2AD67C4843CEB0774EDB029854202EC0 1

        File met2_puf4.txt:2AD67C4843CEB0774EDB029854202EC0 1

        File met2_puf5.txt:2AD67C4843CEB0774EDB029854202EC0 1

        File met2_puf6.txt:2AD67C4843CEB0774EDB029854202EC0 1

        File met2_puf7.txt:2AD67C4843CEB0774EDB029854202EC0 1

        File met2_puf8.txt:2AD67C4843CEB0774EDB029854202EC0 1

        File met2_puf9.txt:2AD67C4843CEB0774EDB029854202EC0 1

        File met2_puf10.txt:2AD67C4843CEB0774EDB029854202EC0 1

Line 34: IDENTICAL for the 33th time
Line 35: IDENTICAL for the 34th time
```

Fig. 8.1: Only one response bit from one file of 10 response bits from 10 files is different

DAD2AFA: For the 1000 CRPs that are present in each file, 997 CRPs of all the 10 files are identical. When the remaining 3 non identical CRPs of 10 files are observed, it is in such a way that for a non identical CRP, only one response bit from one file among 10 response bits of 10 files is different. Remaining 9 response bits are same. It is shown in the Figure 8.2. Just like the Figure 8.2, in all the 3 non identical CRPs only 1 response bit from one file is different. From this it can be said that the implemented PUF is very much accurate in producing results near to ideal expected results. The results can be seen in this document.



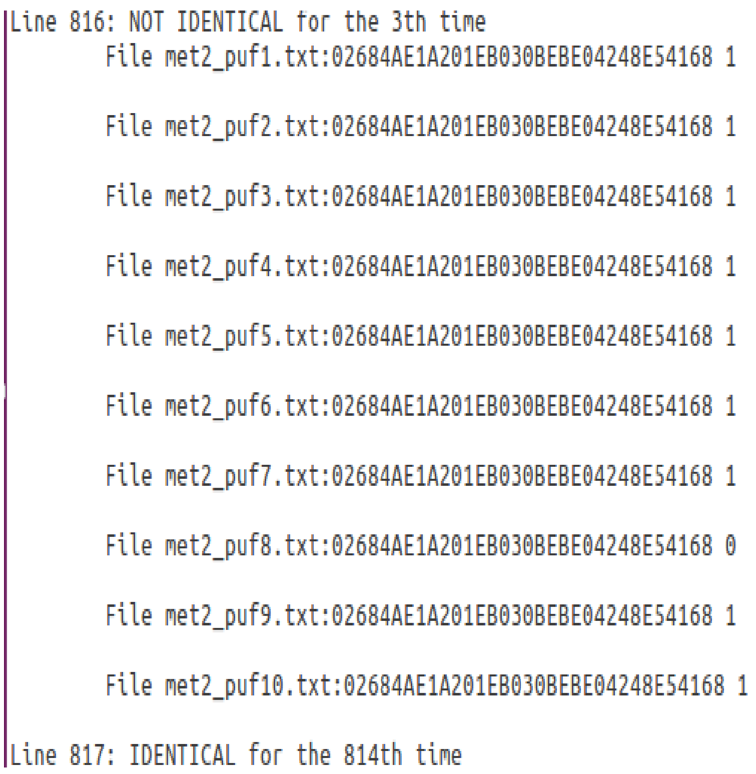
```
Line 272: IDENTICAL for the 272th time
Line 273: NOT IDENTICAL for the 1th time
        File met2_puf1.txt:55585F0B4C91AC76328AFF997087955C 1

        File met2_puf2.txt:55585F0B4C91AC76328AFF997087955C 1

        File met2_puf3.txt:55585F0B4C91AC76328AFF997087955C 1

        File met2_puf4.txt:55585F0B4C91AC76328AFF997087955C 1

        File met2_puf5.txt:55585F0B4C91AC76328AFF997087955C 0

        File met2_puf6.txt:55585F0B4C91AC76328AFF997087955C 1

        File met2_puf7.txt:55585F0B4C91AC76328AFF997087955C 1

        File met2_puf8.txt:55585F0B4C91AC76328AFF997087955C 1

        File met2_puf9.txt:55585F0B4C91AC76328AFF997087955C 1

        File met2_puf10.txt:55585F0B4C91AC76328AFF997087955C 1

Line 274: IDENTICAL for the 273th time
```

Fig. 8.2: Only one response bit from one file of 10 response bits from 10 files is different

DAD2CBB: For the 1000 CRPs that are present in each file, all the 1000 CRPs of all the 10 files are identical. From this it can be said that the implemented PUF accurate in producing results similar to ideal expected results. The results can be seen in this document.

DAD2CF1: For the 1000 CRPs that are present in each file, all the 1000 CRPs of all the 10 files are identical. From this it can be said that the implemented PUF accurate in producing results similar to ideal expected results. The results can be seen in this document.

DAD2BDB: For the 1000 CRPs that are present in each file, 997 CRPs of all the 10 files are identical. When the remaining 3 non identical CRPs of 10 files are observed, it is in such a way that for a non identical CRP, only one response bit from one file among 10 response bits of 10 files is different. Remaining 9 response bits are same. It is shown in the Figure 8.3. Just like the Figure 8.3, in all the 3 non identical CRPs only 1 response bit from one file is different. From this it can be said that the implemented PUF is very much accurate in producing results near to ideal expected results. The results can be seen in this document.

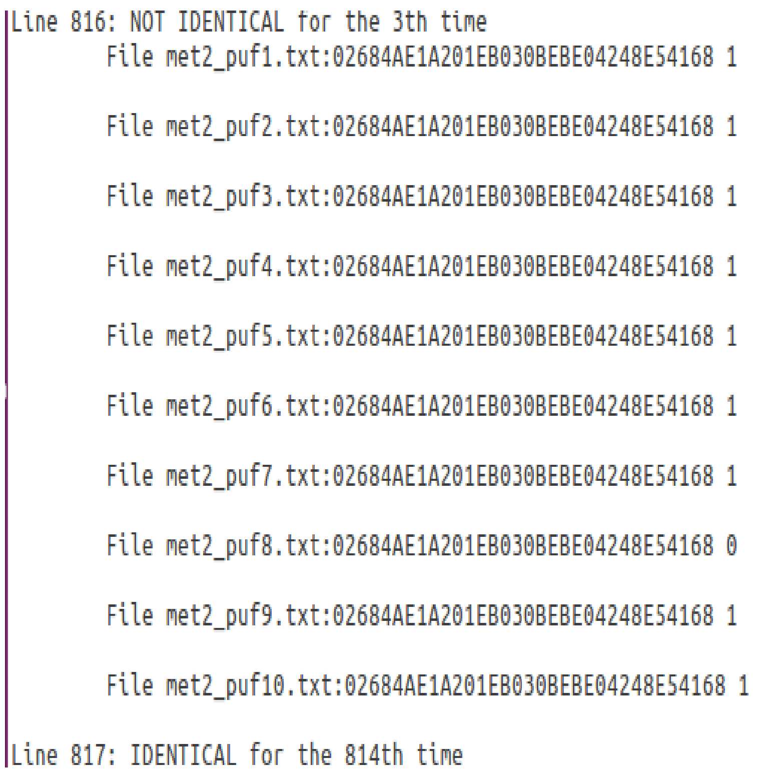


```
Line 816: NOT IDENTICAL for the 3th time
        File met2_puf1.txt:02684AE1A201EB030BEBE04248E54168 1

        File met2_puf2.txt:02684AE1A201EB030BEBE04248E54168 1

        File met2_puf3.txt:02684AE1A201EB030BEBE04248E54168 1

        File met2_puf4.txt:02684AE1A201EB030BEBE04248E54168 1

        File met2_puf5.txt:02684AE1A201EB030BEBE04248E54168 1

        File met2_puf6.txt:02684AE1A201EB030BEBE04248E54168 1

        File met2_puf7.txt:02684AE1A201EB030BEBE04248E54168 1

        File met2_puf8.txt:02684AE1A201EB030BEBE04248E54168 0

        File met2_puf9.txt:02684AE1A201EB030BEBE04248E54168 1

        File met2_puf10.txt:02684AE1A201EB030BEBE04248E54168 1

Line 817: IDENTICAL for the 814th time
```

Fig. 8.3: Only one response bit from one file of 10 response bits from 10 files is different

Expected result: When the same inputs were applied for the different PUFS present on FPGAs, ideally it should provide the different outputs each and everytime.

5 different random text files of the 5 different FPGAs are taken. When the 5 text files are compared, all the 1000 CRPs were non identical which is accurately similar to the expected result. This experiment was done 4 times and everytime the result is same. The results of the 4 times can be seen in document1, document2, document3, document4. It is shown in the Figure 8.4.


```
Line 816: NOT IDENTICAL for the 3th time
        File met2_puf1.txt:02684AE1A201EB030BEBE04248E54168 1

        File met2_puf2.txt:02684AE1A201EB030BEBE04248E54168 1

        File met2_puf3.txt:02684AE1A201EB030BEBE04248E54168 1

        File met2_puf4.txt:02684AE1A201EB030BEBE04248E54168 1

        File met2_puf5.txt:02684AE1A201EB030BEBE04248E54168 1

        File met2_puf6.txt:02684AE1A201EB030BEBE04248E54168 1

        File met2_puf7.txt:02684AE1A201EB030BEBE04248E54168 1

        File met2_puf8.txt:02684AE1A201EB030BEBE04248E54168 0

        File met2_puf9.txt:02684AE1A201EB030BEBE04248E54168 1

        File met2_puf10.txt:02684AE1A201EB030BEBE04248E54168 1

Line 817: IDENTICAL for the 814th time
```
Fig. 8.4: Files from different FPGAs when compared are not identical

The python code used for comparison of files is shown in document.Thus from the above observations the implementation of a 128 bit arbiter PUF is successful.

CHAPTER 9

CONCLUSION AND FUTURE WORK

By the improved PUF based authentication, the device which generates challenges instantly there by decreasing the size of database is more secured from the malicious attacks. And also for the authentication to be more secured the database of the prover and verifier should contain large amount of CRPs which is possible only through a strong PUF. So arbiter PUF which is a strong PUF is chosen for this project. When improved PUF based authentication protocol is performed 93% percent of successful authentications can be observed [26]. In this project a 128 bit arbiter PUF is successfully implemented and the generated CRPs are successfully stored in a text file. Through the tcl code written for this project, a larger number of challenges which may be 1000,10000 or any number can be generated and the respective response of each challenge can be stored in a text file effortlessly. In this PUF the modules or components such as muxes are manually placed to achieve symmetry so the design can be implemented with the minimum routing which means minimum length of wires used. This results in less delay and less leakages which increase in the power efficiency. This made the arbiter PUF more optimised. Thus from the above reasons and results in the chapter 8, the 128 bit arbiter PUF is implemented successfully on a ARTY A7-100 FPGA.

In the **future work** it is proposed to do the implementation on xor PUF. In the future work it is also proposed to implement an arbiter PUF or xor PUF on ASIC.Because on the FPGA we can only do the manual placement but manual routing is not possible. On an ASIC, both manual placement and manual routing are possible. Also ASICs are more power efficient and more faster than FPGAs. These all inclusions make the PUF more optimised. In the future work it is also proposed to perform PUF based protocol between two devices and to be extended for multi devices.

APPENDIX A

Hardware and Software used:

Hardware used:

- FPGA : ARTY A7-100

Software used:

- Vivado for Verilog files
- Vim editor for python files
- Vim editor for tcl files

REFERENCES

1. (). URL `https://www.intrinsic-id.com/sram-puf`.

2. (). **Pranav Kumar Singh**. puf based peer to peer authentication protocols for resource constraint devices.project report, **IIT MADRAS**.

3. **Ashton, K.** *et al.* (2009). That 'internet of things' thing. *RFID journal*, **22**(7), 97–114.

4. **Barbareschi, M.**, **P. Bagnasco**, and **A. Mazzeo** (2015). Authenticating iot devices with physically unclonable functions models. *In 2015 10th International Conference on P2P, Parallel, Grid, Cloud and Internet Computing (3PGCIC)*. IEEE.

5. **Barbareschi, M.**, **A. De Benedictis**, and **N. Mazzocca** (2018). A puf-based hardware mutual authentication protocol. *Journal of Parallel and Distributed Computing*, **119**, 107–120.

6. **Chen, Q.**, **G. Csaba**, **P. Lugli**, **U. Schlichtmann**, and **U. Rührmair** (2011). The bistable ring puf: A new architecture for strong physical unclonable functions. *In 2011 IEEE International Symposium on Hardware-Oriented Security and Trust*. IEEE.

7. **Gassend, B.**, **D. Clarke**, **M. Van Dijk**, and **S. Devadas** (2002*a*). Silicon physical random functions. *In Proceedings of the 9th ACM Conference on Computer and Communications Security*.

8. **Gassend, B.**, **D. Clarke**, **M. Van Dijk**, and **S. Devadas** (2002*b*). Silicon physical random functions. *In Proceedings of the 9th ACM Conference on Computer and Communications Security*.

9. **Gu, C.**, **Y. Cui**, **N. Hanley**, and **M. O'Neill** (2016). Novel lightweight ff-apuf design for fpga. *In 2016 29th IEEE International System-on-Chip Conference (SOCC)*. IEEE.

10. **Gu, C.**, **N. Hanley**, and **M. O'neill** (2017). Improved reliability of fpga-based puf identification generator design. *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, **10**(3), 1–23.

11. **Guajardo, J.**, **S. S. Kumar**, **G.-J. Schrijen**, and **P. Tuyls** (2007). Fpga intrinsic pufs and their use for ip protection. *In International workshop on cryptographic hardware and embedded systems*. Springer.

12. **Guajardo, J.**, **S. S. Kumar**, **G.-J. Schrijen**, and **P. Tuyls** (2008). Brand and ip protection with physical unclonable functions. *In 2008 IEEE International Symposium on Circuits and Systems*. IEEE.

13. **Kodỳtek, F.** and **R. Lórencz** (2015). A design of ring oscillator based puf on fpga. *In 2015 IEEE 18th International Symposium on Design and Diagnostics of Electronic Circuits & Systems*. IEEE.

14. **Kumar, S. S.**, **J. Guajardo**, **R. Maes**, **G.-J. Schrijen**, and **P. Tuyls** (2008). The butterfly puf protecting ip on every fpga. *In 2008 IEEE International Workshop on Hardware-Oriented Security and Trust*. IEEE.

15. **Kursawe, K.**, **A.-R. Sadeghi**, **D. Schellekens**, **B. Skoric**, and **P. Tuyls** (2009). Reconfigurable physical unclonable functions-enabling technology for tamper-resistant storage. *In 2009 IEEE International Workshop on Hardware-Oriented Security and Trust*. IEEE.

16. **Lim, D.**, **J. W. Lee**, **B. Gassend**, **G. E. Suh**, **M. Van Dijk**, and **S. Devadas** (2005). Extracting secret keys from integrated circuits. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, **13**(10), 1200–1205.

17. **Maes, R.**, **P. Tuyls**, and **I. Verbauwhede** (2008). Intrinsic pufs from flip-flops on reconfigurable devices. *In 3rd Benelux workshop on information and system security (WISSec 2008)*, volume 17.

18. **Maiti, A.**, **V. Gunreddy**, and **P. Schaumont** (2013). A systematic method to evaluate and compare the performance of physical unclonable functions. *In Embedded systems design with FPGAs*, 245–267. Springer.

19. **Maiti, A.** and **P. Schaumont** (2012). A novel microprocessor-intrinsic physical unclonable function. *In 22nd International Conference on Field Programmable Logic and Applications (FPL)*. IEEE.

20. **Pappu, R.**, **B. Recht**, **J. Taylor**, and **N. Gershenfeld** (2002*a*). Physical one-way functions. *Science*, **297**(5589), 2026–2030.

21. **Pappu, R.**, **B. Recht**, **J. Taylor**, and **N. Gershenfeld** (2002*b*). Physical one-way functions. *Science*, **297**(5589), 2026–2030.

22. **Rührmair, U.**, **J. Sölter**, **F. Sehnke**, **X. Xu**, **A. Mahmoud**, **V. Stoyanova**, **G. Dror**, **J. Schmidhuber**, **W. Burleson**, and **S. Devadas** (2013). Puf modeling attacks on simulated and silicon data. *IEEE transactions on information forensics and security*, **8**(11), 1876–1891.

23. **Simons, P.**, **E. van der Sluis**, and **V. van der Leest** (2012). Buskeeper pufs, a promising alternative to d flip-flop pufs. *In 2012 IEEE International Symposium on Hardware-Oriented Security and Trust*. IEEE.

24. **Su, Y.**, **J. Holleman**, and **B. P. Otis** (2008). A digital 1.6 pj/bit chip identification circuit using process variations. *IEEE Journal of Solid-State Circuits*, **43**(1), 69–77.

25. **Yu, H.**, **P. H. Leong**, and **Q. Xu** (2011). An fpga chip identification generator using configurable ring oscillators. *IEEE transactions on very large scale integration (VLSI) systems*, **20**(12), 2198–2207.