ABSTRACT

Title of Document:	SYNERGISTIC CONFIGURABLE RING OSCILLATOR PUF: DESIGN, CHARACTERIZATION, AND IMPLEMENTATION
	Khai Vinh Lai, Master of Science, 2014
Directed By:	Professor, Dr. Gang Qu Department of Electrical & Computer Engineering

Silicon Physical Unclonable Function (PUF) is a novel hardware primitive that uses the intrinsic variation of integrated circuit manufacturing process for various security applications. Ring oscillator PUF (RO PUF) is one of the most popular silicon PUFs due to its ease of implementation on both ASIC and FPGA. However, RO PUF can have severe reliability issues when the operating environment deviates from the normal environment and security issues when it lacks an efficient anti-cloning mechanism. In this work, we propose a novel approach to build reliable RO PUF efficiently and enhance its resistance against physical cloning attack. The key idea of our approach is to construct ring oscillators with carefully selected inverters during the testing phase after the chip is fabricated. Our experimental results show that our configurable approach outperforms the traditional RO PUF and 1-out-of-8 PUF by generating more reliable bits that pass the NIST randomness tests. Our approach is

also more hardware efficient than these RO PUFs. We also demonstrate that the configuration vectors can prevent physical cloning and have the potential usage in chip-dependent applications such as device authentication.

SYNERGISTIC CONFIGURABLE RING OSCILLATOR PUF: DESIGN, CHARACTERIZATION, AND IMPLEMENTATION

By

Khai Vinh Lai

Thesis submitted to the Faculty of the Graduate School of the
University of Maryland, College Park, in partial fulfillment
of the requirements for the degree of
Master of Science
2014

Advisory Committee:
Professor Dr. Gang Qu, Chair
Professor Dr. Charles Silio
Professor Dr. Charalampos Papamanthou

Dedication

To my parents who always love and support me unconditionally. To my grandmother who is forever be present in my heart and thought. To my aunt and cousins in the US whose wholehearted love has helped me overcome obstacles in life. And to my wife who is always there for me.

Acknowledgements

First of all, I would like to express my deepest gratitude to my advisor, Dr. Gang Qu, who supported my graduate study, introduced me to the exciting field of Physical Unclonable Function (PUF), taught me the rigorous scientific process of research, and provided valuable insights and criticisms to my thesis. I would also like to thank my dear friend Mingze Gao for his numerous clever suggestions during our daily discussions, his valuable contribution to our published paper, and most importantly his enthusiasm in conducting research that motivated me to work extra hard. I would also like to extend my gratitude to Dr. Chi-En Yin who made available the source code of his pioneer work in ring oscillator PUF so that I can build my work upon it. I want to express my appreciation to Jiliang Zhang for his feedbacks of my work. I also want to greatly thank my fellow graduate friend Tanvir Arafin for his suggestion about how to build a temperature-controlled chamber. I also want to extend my appreciation to Dr. Patrick Schaumont and his team at Virginia Tech for sharing their large-scale experimental result in ring oscillator PUF, which is essential to my simulation work.

Lastly, I would like to thank my committee members for their service, the ECE faculty for teaching me the skills and knowledge necessary for my future career, and the ECE Department for their unparalleled support during my educational journey at the University of Maryland.

Table of Contents

Dedication ii
Acknowledgements iii
Table of Contents iv
List of Tables vii
List of Figures viii
1 Introduction 1
1.1 Introduction to Silicon PUF 1
1.2 Silicon PUF Taxonomy 2
1.2.1 Analog Electronic PUFs 3
1.2.2 Memory-based PUFs 4
1.2.3 Delay-based PUFs 6
1.3 Thesis Organization 7
2 Literature Review 9
2.1 Improving PUF's Reliability 9
2.2 Configurability and Reconfigurability 11
3 Motivation Example 12
3.1 Example 1 12
3.2 Example 2 13
4 Configurable RO PUF 17
4.1 Architecture of Configurable RO PUF 17
4.2 Inverter Delay Measurement 18
4.3 Design of the Configurable RO PUF 20
4.4 Solving the Inverter Selection Problem 21
4.4.1 Case-1: Both ROs Uses the Same Configuration Vector 22

4.4.2 Case-2: ROs Uses Different Configuration Vector 22
4.5 Configurable One-RO-One-Bit (OROB) PUF 24
4.6 Authentication Using Configuration Vector 26
4.7 Anti-cloning Using Bistable Ring 26
5 Experimental Results 27
5.1 Simulation 27
5.1.1 Randomness of Configurable PUF's Output 28
5.1.2 Uniqueness of Configurable PUF's Output 29
5.1.3 RO Configuration Information 31
5.1.4 Reliability 34
5.1.5 Uniqueness of OROB Approach 40
5.1.6 Reliability of OROB Approach 41
5.1.7 Dual Voltage Scaling Anti-Cloning Scheme 42
5.2 FPGA Implementation 44
5.2.1 Overview of FPGA Implementation 44
5.2.2 Measurement and Analysis Procedure 48
5.2.3 Uniqueness Result 49
5.2.4 Reliability Result 50
6 Conclusion 55
7 Appendices 56
7.1 Verilog Code 56
7.1.1 top.v 56
7.1.2 arith_shift_right_generic.v 62
7.1.3 controller.v 63
7.1.4 uart_transceiver.v 76
7.1.5 puf.v 79

7.1.6 frd.v 81
7.1.7 cfg_ro.v 82
7.1.8 ripple_counter_generic.v 83
7.1.9 global_defs.verh 84
7.2 C Code 85
7.2.1 main.cpp 85
7.2.2 defs.h 105
Bibliography 108

List of Tables

Table 1. NIST TEST RESULTS OF CONFIGURABLE PUFS' OUTPUTS FOR CASE-1. 29

Table 2. NIST TEST RESULTS OF CONFIGURABLE PUFS' OUTPUTS FOR CASE-2. 29

Table 3. TOTAL NUMBER OF BITS PER BOARD 36

Table 4. PUF BITS PER BOARD (OUR OROB APPROACH VERSUS OTHER RO PUFS) 42

Table 5. RELIABILITY OF OUR OROB APPROACH VERSUS OTHER RO PUFS 42

Table 6. DUAL-VOLTAGE FLIPPING PATTERN 44

Table 7. TOTAL NUMBER OF PUF RESPONSE BITS FROM 9 FPGA BOARDS 51

List of Figures

Figure 1. Silicon PUF Taxonomy 3

Figure 2. SRAM PUF: Cross-Coupled Inverters [11] 5

Figure 3. Butterfly PUF [11] 5

Figure 4. Arbiter PUF [17] 6

Figure 5. RO PUF [17] 7

Figure 6. Architecture of Configurable RO PUF 18

Figure 7. One Delay Unit in Configurable RO PUF 18

Figure 8. 2:1 MUX 19

Figure 9. Measurement Circuit 19

Figure 10. Inter-Chip HD of Configurable PUF Outputs for Case 1 30

Figure 11. Inter-Chip HD of Configurable PUF Outputs for Case 2 31

Figure 12. Intra-Chip HD of Best Configuration For Case-1 33

Figure 13. Intra-Chip HD of Best Configuration For Case-2 33

Figure 14. Percentage of Bit Flips under Voltage Variation for Case 1 35

Figure 15. Percentage of Bit Flips under Temperature Variation for Case 1 38

Figure 16. Percentage of Bit Flips under Voltage Variation for Case 2 39

Figure 17. Percentage of Bit Flips under Temperature Variation for Case 2 40

Figure 18. Inter-Chip HD of Configurable OROB PUF Outputs 41

Figure 19. Homemade Temperature Controlled Chamber 45

Figure 20. FPGA Implementation of Configurable RO PUFs 46

Figure 21. Placement of Configurable RO PUF & Measurement Flip-Flops in Virtex-5 47

Figure 22. Inter-Chip HD of Configurable PUF Outputs on 9 FPGA boards..... 50

Figure 23. Reliability Comparison of Our Configurable Approach to Other RO PUFs 53

Figure 24. FPGA Hardware Utilization Comparison 54

1 Introduction

1.1 Introduction to Silicon PUF

Since the first transistor was invented at Bell Labs in 1947, there has been an explosion in new inventions and innovations that facilitate economic growths and improve the lives of billions of people. Nowadays electronic products have become so ubiquitous and essential in our daily lives that there is an increasing demand in making them more secured to protect confidential information and intellectual properties against any attacker.

It is well-known that a skilled attacker can use advanced invasive physical attacks – such as micro-probing, laser cutting, selective etching, glitch attacks, power analysis, etc. – to remove a package and reconstruct the layout of a circuit or even read out the secret information stored in non-volatile memories [1] [2]. Although there exists tamper-sensing techniques like sensor mesh [2] to hinder invasive physical attacks, such techniques require power to operate and are costly to implement.

Silicon Physical Unclonable Function (PUF) [3] [4] [5] is an innovative circuitry that can extract the intrinsic physical characteristics of an integrated circuit (IC) like signal propagation delay and threshold voltage – such characteristics are unique to each IC due the random variation in fabrication process – and use such unpredictable intrinsic information for security applications. Compared to existing tamper-sensing techniques, PUF offers higher protection against invasive physical

attacks without requiring power to operate because any physical tampering of an IC, regardless of the power being on or off, will guarantee to permanently change the IC's physical characteristics and therefore corrupt the stored secret information. Unlike the sensor mesh, PUF does not require any extra fabrication step because it can be implemented with common digital logics, so its fabrication cost is lower.

Applications of PUF can be divided into 3 categories: system identification, secret key generation, and hardware entangled cryptography [6]. For system identification, PUF can be used for anti-counterfeiting, hardware binding, and hardware metering. For secret key generation, PUF can be used for secure key storage and secure key distribution where a secret key is embedded in the complex physical characteristics of an IC. For hardware entangled cryptography, PUF can be integrated into crypto primitives where secret parameters are device-dependent.

1.2 Silicon PUF Taxonomy

Figure 1 shows three categories of silicon PUF and existing PUFs belong to each category. In this section, we will briefly discuss each category and its corresponding PUFs.

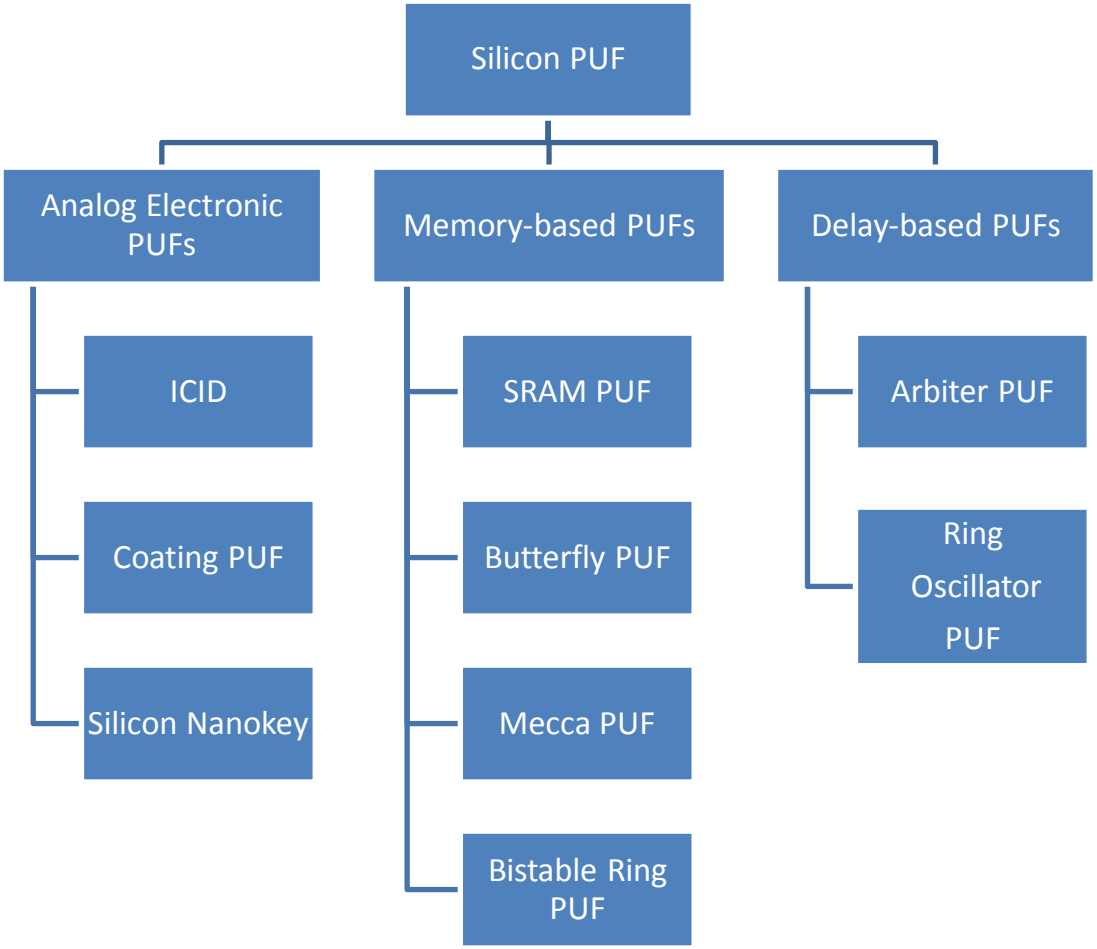

Figure 1. Silicon PUF Taxonomy

1.2.1 Analog Electronic PUFs

Analog electronic PUFs extract the random variation in drain current, capacitance, or threshold voltage. For example, Lofstrom et al. [7] measures the

variation in drain current and use it to create IC identification (ICID). Tuyls et al. [8] covers an IC with an opaque and chemically inert coating doped with random dielectric particles to protect it from invasive physical attacks. There is an array of sensors locating at the top metal lay of the IC to measure the capacitance of each local point. An electrical signal of a certain frequency and amplitude applied at a certain point serves as a challenge, and the measured capacitance serves a response which is then converted into a key or an ID. Puntin et al. [9] uses the variation in minimum size MOSFET's threshold voltage to create a silicon nanokey.

1.2.2 Memory-based PUFs

Memory-based PUFs harvests the uninitialized values from memory circuits to generate a key or an ID. For example, SRAM PUF [10] [11] consists of an array of memory units. Each memory unit is formed by two cross-coupled inverters as shown in Figure 2 and will settle down to 0 or 1 after power on. The settle-down value is determined by the variation in the intrinsic physical characteristics of the inverters. The selection of a subset of memory units whose values are read after power on serves as a challenge, and the read-out values serves as a response. Because not all FPGAs support uninitialized memories, SRAM PUF is not suitable for every FPGA implementation. Kumar et al. [12] proposes an improvement to SRAM PUF by replacing the inverters with latches or flip-flops as shown in Figure 3 – this is called Butterfly PUF and is suitable for all types of FPGA. Krishna et al. [13] proposes MECCA PUF to defeat the exhaustive read attack in SRAM PUF and Butterfly PUF by using the write pulse duration as a challenge and the write failure rate as a

response; however, MECCA PUF is only suitable for embedded memory of specific structures [14]. Chen et al. [15] proposes Bistable Ring PUF that can produce an exponential number of challenge-response pairs by selecting different combinations of inverters from a loop of even number of inverters.

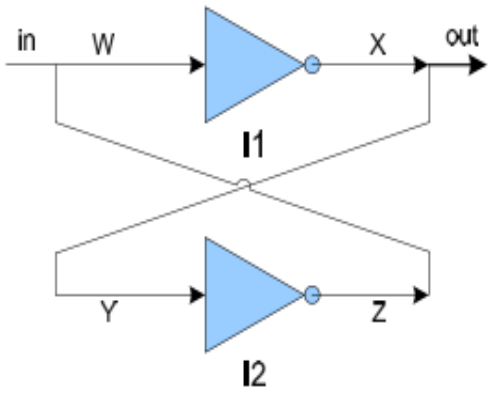

Figure 2. SRAM PUF: Cross-Coupled Inverters [11]

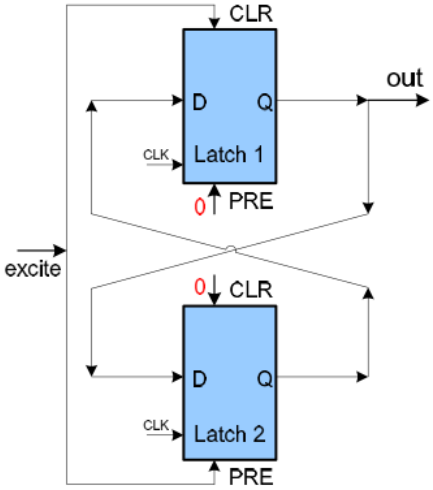

Figure 3. Butterfly PUF [11]

1.2.3 Delay-based PUFs

Delay-based PUFs extracts the variation in signal propagation delay and digitize it into chip ID or secret key through comparison. The two most popular delay-based PUFs are Arbiter PUF and Ring Oscillator (RO) PUF.

First introduced in [5] [16], an Arbiter PUF consists of 128 stages as shown in Figure 4. Each stage is made of two multiplexers in parallel whose inputs are shared and outputs go to the next stage. A step signal is sent to the first stage's input and propagated through a path pre-selected by a 128-bit challenge (x[0] ~ x[127]). Ideally, the step signal should arrive at the D input and clock input of the latch at the end of the chain at the same time. But in the presence of process variation, one path can have more propagation delay than the other and thus a logic-1 will be latched to the output Q if the step signal arrives at the D input before it arrives at the clock input. Conversely a logic-0 will be latched to the output if the step signal arrives at the clock input before it arrives at the D input. This latched value can be used as a 1-bit PUF signature or a response to a challenge.

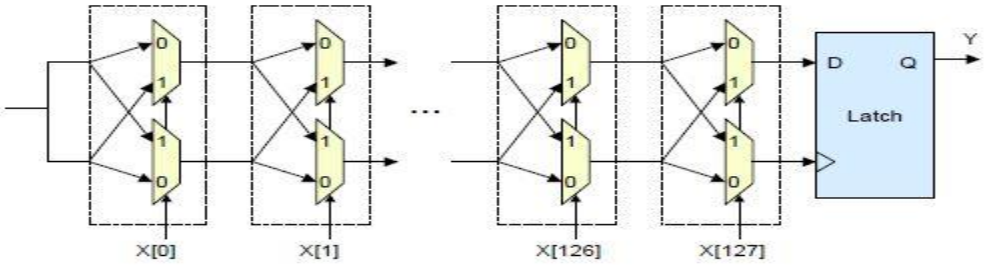

Figure 4. Arbiter PUF [17]

Although the Arbiter PUF is small and fast, it has two main drawbacks. First, it requires symmetrical routing both inside and outside of each stage; otherwise, the

uniqueness of the responses is reduced. This makes the implementation of Arbiter PUF on FPGA difficult [18]. Second, Arbiter PUF can be vulnerable to model-building attack assisted by machine learning as reported in [19].

Ring Oscillator PUF, whose structure is shown in Figure 5, was first proposed in [17]. The PUF consists of a pair of ring oscillators whose frequencies are measured and then compared to produce a logic-1 if the top ring is faster than the bottom ring; otherwise a logic-0 is produced. Compared to Arbiter PUF, RO PUF is more relaxed on the requirement of routing symmetry and so it can be easily implemented on FPGA. However, RO PUF is bigger, slower, and consumes more power and therefore too costly for resource constrained applications like RFID [17].

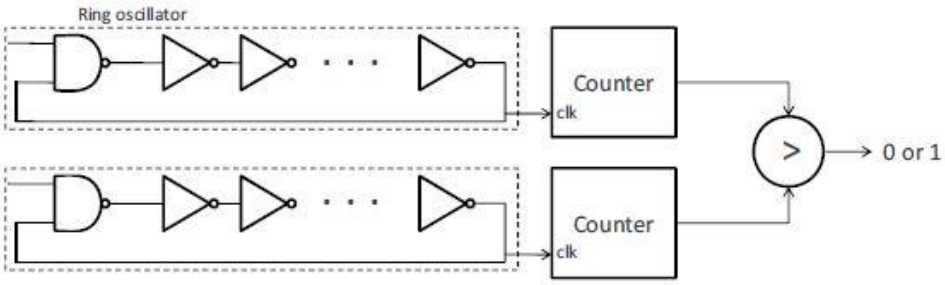

Figure 5. RO PUF [17]

1.3 Thesis Organization

This thesis is organized as follows. Chapter 2 gives an overview of the state-of-the-art designs of RO PUF and makes a distinction between our design and these designs. Chapter 3 gives two motivation examples on reliability and cloning issues. Chapter 4 talks in-depth about the design of our configurable RO PUF. Chapter 5

presents the simulation and FPGA-implementation results of our design. Chapter 6 gives the conclusion of this study and future works. In the appendices, the Verilog code of our design and the C code of the communication process are presented.

2 Literature Review

Due to the scope of this study, we focus our literature review on existing designs of RO PUF for reliability and hardware efficiency. We also review previous studies on configurable and reconfigurable PUFs and make a distinction between our approach and these state-of-the-art approaches.

2.1 Improving PUF's Reliability

Since the inception of RO PUF, there have been several proposals on how to improve its reliability. Suh and Devadas, the inventors of RO PUF, first suggest the 1-out-of-8 scheme where the fastest and slowest rings are picked among 8 ring oscillators to form the most reliable pair [17]. This scheme can achieve 0.48% PUF output bit flips in the presence of temperature and supply voltage variations. But its hardware cost is 4 times greater than the original RO PUF which uses only two ROs to generate one PUF bit [17]. Yin and Qu [20] proposes a temperature-aware cooperative scheme that can achieve 80% higher hardware efficiency than the 1-out-of-8 scheme, but this scheme requires an on-chip temperature sensor. Vivekraja and Nazhandali [21] propose a scheme to operate the RO PUF at different optimal voltages depending on the temperature so that the least number of bit flips occur. This approach can achieve from 90% to 100% hardware utilization. It also improves the traditional RO PUF's reliability by 19% for temperature variation and 14% for voltage variation. However, this approach also requires an on-chip temperature sensor. Most recently, Mansouri and Dubrova [22] proposes an improvement to

Vivekraja and Nazhandali's scheme by applying different voltages on different inverters in the same ring oscillator. This approach also requires specialized circuits which can increase the hardware overhead. Yin and Qu [23] propose to compare a group of ROs instead of only one pair and at the same time generate multiple bits. This group-based approach can improve reliability, maximize the number of PUF bits, and improve hardware efficient.

Another approach to increase PUF reliability is to use error correction codes to detect and correct the PUF bits that are flipped. Yu and Devadas [24] proposes a new syndrome coding scheme, called Index-Based Syndrome Coding, that leaks less information than conventional syndrome coding methods and can achieve zero error correction failure per million trials. Dodis et al. [25] proposes two primitives: the fuzzy extractor and the secure sketch. The fuzzy extractor can extract a nearly uniform randomness R from an input source and R will be the same for every extraction as long as the change in the input source is small. The secure sketch on the other hand can provide public information about an input without revealing the input. Although these approaches are effective in increasing the PUF's reliability, they introduce additional hardware cost due to the error correction circuits.

Our approach on the other hand doesn't require an on-chip temperature sensor, or operation at different supply voltages, or extra complex circuits to increase the PUF's reliability. Instead, we achieve high reliability by maximizing the delay difference between two ROs.

2.2 Configurability and Reconfigurability

Maiti and Schaumont [26] introduce the notion of configurability in RO PUF. In their approach, there is a multiplexer at each stage of a RO to select one out of two inverters. For a RO with three stages (i.e., three inverters), there are eight possible configurations and the configuration that provides the largest delay difference will be used to generate one PUF output bit. The implementation of their 3-stage RO PUF on a Xilinx FPGA occupies 2 Configurable Logic Blocks (CLB). Taking advantage of the CLB's internal architecture, Xin et al. [27] increase the number of possible configurations to 256 while still using the same number of CLBs as Maiti and Schaumont's implementation. Compared to these approaches, our approach does not depend on the FPGA architecture and is more flexible because at any stage of the RO we have the option of not using an inverter. This fact makes our configurable RO PUF fundamentally different from their approaches.

There are several studies on reconfigurable PUFs like [28] [29]. They propose mechanisms where a PUF can be transformed into a new PUF with unpredictable behaviors without physically replacing or modifying the PUF. And this allows the secret key or cryptographic primitives based on PUFs to be updated. Our approach is completely different from these approaches because, in our approach, once a RO PUF is configured, it will remain unchanged and cannot be reconfigured.

3 Motivation Example

3.1 Example 1

Consider a pair of ring oscillators, RO1 and RO2, each consisting of 7 inverters. Assuming that the delays of these inverters are: {d_1=5, d_2=4, d_3=8, d_4=7, d_5=4, d_6=6, d_7=3} and {c_1=7, c_2=5, c_3=7, c_4=5, c_5=1, c_6=4, c_7=5}, where d_i and c_i denote the delay of the i^{th} inverter in RO1 and RO2, respectively. The total delay of these two ROs can be computed as

$$D_{RO1} = 5+4+8+7+4+6+3=37$$

$$D_{RO2} = 7+5+7+5+1+4+5=34$$

RO1 is 37-34 = 3 units of time slower than RO2. This delay difference can be used to generate one PUF bit and its reliability is determined by this delay difference. In general, a large delay difference leads to a reliable bit. Consider selecting {d_4, d_5, d_6} in RO1 and {c_4, c_5, c_6} in RO2, respectively. The delay difference will be (7+4+6)-(5+1+4) = 7 units, which is more than twice as large as the delay difference when all the inverters are included in the ROs.

The rationale behind this is that the fabrication variation is unpredictable. An inverter in RO1 is equally likely to be faster or slower than the inverter at the same position in RO2. When all the inverters are included in the ROs, the delay difference between inverters at the same position can be canceled out. In this example, although RO1 is slower than RO2, it has faster inverters at the first, second and last position than RO2, including these three inverters in the ROs will reduce their delay

difference. When we selectively choose inverters to configure the RO, we can increase the gap between two ROs' total delays and thus improve the PUF bit's reliability.

Furthermore, let us consider the 3 fastest inverters with the 3 slowest inverters in each RO. For RO1, the total delay of the 3 fastest ones D_{fast_1} = 4+4+3 =11 and the total delay for the 3 slowest ones is D_{slow_1} = 8+7+6 =21. We see a delay difference of 10 units of time. For RO2, we have D_{fast_2} = 4+1+5 =10, D_{slow_2} = 7+7+5 =19, and a delay gap of 9 units of time. If we can create a PUF bit by comparing the faster inverters with the slower inverters in the same RO, we are able to generate bits from both RO1 and RO2 that are more reliable than the bit generated by comparing RO1 and RO2. More important, we get two PUF bits instead of one.

3.2 Example 2

PUF is short for "Physical Unclonable Function", and obviously the most important feature is "Unclonable". However, this feature is threatened as attack techniques become more sophisticated. To see how PUF becomes clonable, let us review the short history of PUF. Silicon PUFs are based on unclonable intrinsic fabrication variation in delay, capacitance, or threshold voltage. First of all, when we extract such analog signals and map them into binary 0's and 1's, we cannot claim that the binary value (1 or 0) and the analog signals are equally unclonable. For example, it is infeasible to build two ROs with exactly the same delay. However, given a pair of ROs and we randomly fabricate another pair, if we believe the intrinsic fabrication variation is truly random, then the probability that the new pair generates

the same bit as the given pair is 50%. Such a probability that is too large to be claimed unclonable. However, this is acceptable because an RO PUF consists of hundreds of RO pairs and the coincidence that two PUFs generating exactly the same PUF bits will be small enough to be deemed unclonable as long as it remains infeasible to measure the delay of a single RO (or the delay difference between a given pair of ROs).

This leads us to the second place where the unclonable feature is lost or weakened. When the intrinsic fabrication variations are not large enough to generate reliable PUF information, various methods are proposed to physically enlarge the variation and thus improve the reliability. For example, optical proximity correction method is used make PUF information unique [30], and several approaches have been proposed to improve the reliability of delay-based PUF under temperature variations [31]. As a side effect, these techniques make it possible to physically extract the variation that is used to generate PUF bits. It will be more effective to use EM emanation approach to capture the frequency differences if these frequency values are not very close. When the underlying fabrication variation is available, cloning the PUF becomes simple. In the case of SRAM PUF, once each SRAM cell is characterized, it can be cloned by standard university failure analysis equipment [32]. In the case of RO PUF, it is trivial to build a RO faster than the other (for example, by adjusting gate size or threshold voltage or simply replacing even number of inverters with wires).

Here is a simple example to prove the feasibility of cloning RO PUF. There are two pairs of ROs {A, B} and {C, D}, with 5 inverters in each ring, generating two bits 01. This means delay(A) > delay(B), and delay(C) < delay(D). The attackers can use the EM emanation to measure these two relations. To clone this PUF, they can simply build A and D with 5 inverters and build B and C with 1 inverter. The inverter numbers' mismatching will guarantee that the cloned PUF generates the same response as the original one. So the lesson we have learned here is that the intrinsic fabrication variation that silicon PUFs based on is unclonable, it is during the process of generating secure and reliable PUF information that we make these variations measureable and hence PUF information clonable.

The following motivation example will illustrate how our configurable RO PUF can use a voltage scaling scheme to detect and thus defeat physical cloning. Consider the following two ROs, each consisting of 5 inverters whose delays are:

At voltage V1: RO1={4, 5, 3, 7, 6}

RO2={6, 3, 4, 7, 4}

At voltage V2 (V2<V1):

RO1={10, 12, 9, 13, 11}

RO2={11, 8, 10, 12, 9}

As the supply voltage decreases, the delays of the inverters increase. Because of the configurable feature, we don't have to use all the inverters; in this example, we can choose the first three. At V1, delay(RO1)=4+5+3=12, delay(RO2)=6+3+4=13, and delay(RO1) < delay(RO2). But when we change the voltage to V2,

delay(RO1)=10+12+9=31, delay(RO2)=11+8+10=29, and so delay(RO1) > delay(RO2). So if at V1 the PUF output bit is “0”, the bit will flip to “1” when the supply voltage changes to V2. Prior to the PUF secret re-generation phase, we can have a PUF authenticity verification phase where we check if the PUF is a physical clone by setting the configuration vectors of RO1 and RO2 to {11100}, running the PUF at voltage V1 and generating the response bit-stream S1. Then we change the voltage to V2 through a power management unit and get another response bit-stream S2. If S2 matches the complement of S1, the PUF’s authenticity is confirmed. Moreover, because the usages are authenticity and prevent cloning, the controlled flipping bits {S1, S2} do not need to be exactly the same. Majority match is acceptable.

4 Configurable RO PUF

4.1 Architecture of Configurable RO PUF

As we have shown in the motivational example, we want to select inverters to configure the ROs based on the delay information of the inverters. This requires both the measurement of inverter delay and a flexible architecture that allows us to do the configuration. We address these concerns in this and the next section. Figure 6 depicts the architecture that gives us the flexibility to select inverters for the construction of ROs. A multiplexer will be added after each inverter to control whether the inverter will be included in the RO. This is achieved by the selection bit of the multiplexer. If the selection bit is "1", the corresponding inverter will be included in the RO; if the selection bit is "0", the corresponding inverter will not be included in the RO and the signal will go through the wire to the next inverter. The term configuration vector is used to refer to the collection of all the multiplexer selection bits.

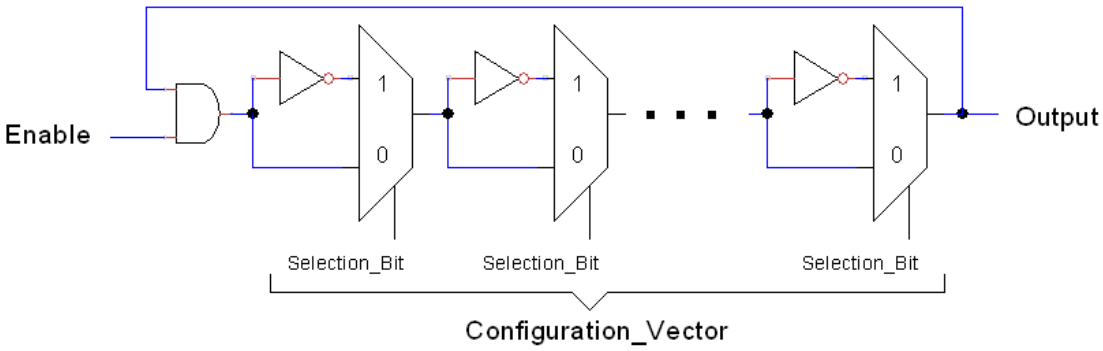

Figure 6. Architecture of Configurable RO PUF

4.2 Inverter Delay Measurement

In practice, it is very difficult to measure the delay of single inverter because it oscillates very fast. However, as our design is very flexible, we don't need to measure the single inverter's delay individually. We can send different configurations and calculate the delay by the existing measurement data.

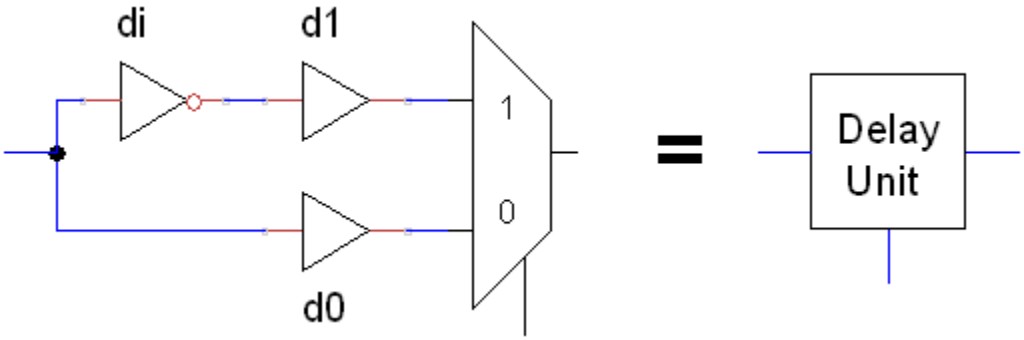

Figure 7. One Delay Unit in Configurable RO PUF

The main unit is demonstrated in Figure 7. We consider the left structure as one delay unit. As the PUF is very sensitive to the process variation, we need to concern about all the potential variation in our design. Let define the delay of path-1 as the path delay from point x1 to point y, and respectively the delay of path-0 as the path delay from point x0 to point y in Figure 8.

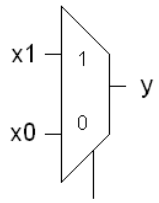

Figure 8. 2:1 MUX

Because, in real circuits, multiplexers are never ideal, the delay of path-1 and delay of path-0 should not be exactly identical. We can model these two delays as two buffers and pull them out of the multiplexer, as illustrated in Figure 7. We denote d1 as the delay of path-1, d0 as the delay of path-0, and di as the delay of an inverter. After we pull the delays out of the multiplexer, the multiplexer becomes "ideal" (does not contain any delay). We first measure di+d1 by setting the selection bit to 1, and measure d0 by setting the selection bit to 0. Then we calculate the delay difference: ddiff=di+d1-d0. So we can treat the delay of a Delay Unit as ddiff.

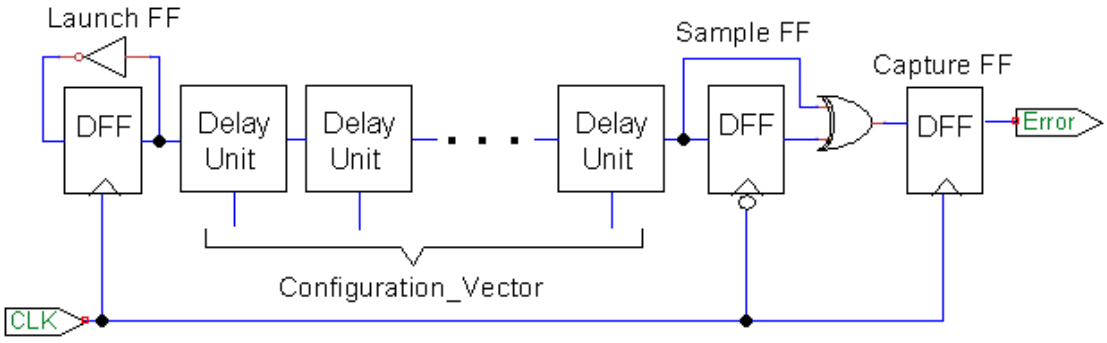

Figure 9. Measurement Circuit

Figure 9 provides our proposed measurement circuit, which is inspired by the approach in [33] and [34]. To measure the delay of the j^{th} Delay Unit, we first configure all the Delay Units to use path-0, then configure only the j^{th} Delay Unit to

use path-1, and then find their delay difference. By sweeping the clock signal from low frequency to high frequency, we can obtain the probability distribution of error and estimate the delay of a Delay Unit at 50% error rate [33] [34]. In our design, what we concern is only the delay difference. So for every Delay Unit, if we set the multiplexer's selection bit to 1, the delay of that unit is ddiff, otherwise the delay is 0. Our approach does not require a very high accuracy of the measurement because we do not need the actual delay information of each inverter as long as we can tell their relative speed (which one is faster and which one is slower).

4.3 Design of the Configurable RO PUF

We deploy the configurable ROs in pairs on chip with n inverters in each RO. After the chip is fabricated and during the chip testing phase, we measure the delay of each inverter as mentioned in the previous section. Next, based on these measurements, we determine the configuration vector for each RO pair to configure them. Each configured RO pair will generate one PUF bit.

Our goal is to generate PUF output bits that are (1) reliable under different operating environments, (2) random so they will be robust against potential attacks, (3) unique so different chips will unlikely generate the same output bits. Our proposed configurable RO PUF has advantages in achieving these goals over the existing RO PUF design approaches.

First, we configure the ROs after chip is fabricated. This allows us to measure the real delay information to construct ROs. Existing RO PUF just uses whatever variation comes during the fabrication process. Second, we only put selective

inverters in the RO, while current RO PUF uses all the inverters. We have seen from the motivational example that this can significantly increase the delay difference between a pair of ROs and thus make the PUF bit more reliable. Third, when we cannot find a subset of inverters to generate a large delay difference between a pair of ROs, we don't have to use the PUF bit generated from this pair. This can eliminate the cost of error correction circuitry.

A major challenge in the proposed configurable RO PUF is how to find the subset of inverters that can maximize the delay difference between a pair of RO. If there are n inverters for each RO, then there are $\binom{n}{1} + \binom{n}{3} + \cdots + \binom{n}{k} + \cdots + \binom{n}{n}$ (assuming that n is odd) possible configurations. It will be expensive to evaluate each option, particularly when n is large. In the next subsection, we present our solution to this question.

4.4 Solving the Inverter Selection Problem

For a pair of ROs, we refer to them as top RO and bottom RO. The inverter selection problem aims to find configuration for the two ROs (that is, which inverter will be selected to construct the RO) such that their delay difference will be maximized. We assume that both RO will use the same number of inverters once configured. This is for security concern because the one that used fewer inverters will be faster most of the time, making it easier for an attacker to guess the bit value this pair of ROs generates. We consider two different cases: Case-1 is when both ROs use the same configuration vector; and Case-2 is when they have different configurations.

4.4.1 Case-1: Both ROs Uses the Same Configuration Vector

Let ($x_1\ x_2\ \ldots\ x_n$) and ($y_1\ y_2\ \ldots\ y_n$) be the configuration vector for the top RO and bottom RO, and α_i and $ß_i$ be the delay of the i^{th} inverter in the top RO and bottom RO, respectively. In Case-1, because we require the top RO and the bottom RO to have the same configuration $x_i = y_i\ \forall i = 1, \ldots, n$, so we need to determine the value for each x_i such that

$$\arg\max_{\mathbf{x}} \left| \sum_{i=1}^{n} \Delta d_i \right| \ , \ \text{where} \ \ \Delta d_i = (\alpha_i - \beta_i) \cdot x_i \ \ \text{and} \ \left(\sum_{i=1}^{n} x_i \right) = \left(\sum_{i=1}^{n} y_i \right) \text{is odd} \qquad (1)$$

Equation (1) can be solved by realizing that the absolute summation is the largest if all of the terms Δd_i are having the same sign. Let $\Delta+$ be the sum of all the Δd_i's with positive values and $\Delta-$ be the sum of all the Δd_i's with negative values (we can ignore those zero-valued Δd_i's). If $\Delta+ > -\Delta-$, we select the i^{th} inverter for both top and bottom ROs whenever $\Delta d_i > 0$; otherwise, we select the i^{th} inverter for both top and bottom ROs whenever $\Delta d_i < 0$. This is clearly the optimal solution for the inverter selection problem (1).

4.4.2 Case-2: ROs Uses Different Configuration Vector

When the top RO and bottom RO do not have to have the same configuration, we denote $\alpha_{(i)}$ and $ß_{(i)}$ as the i^{th} largest inverter delay value of the top RO and the bottom RO, respectively. That is, $\alpha_{(1)}=\max(\alpha_1, \alpha_2,\ldots,\alpha_n)$, $\alpha_{(2)}$ is the second largest among $(\alpha_1, \alpha_2,\ldots,\alpha_n)$, and $\alpha_{(n)}=\min(\alpha_1, \alpha_2,\ldots,\alpha_n)$. Let $x_{(i)}=x_j$ where j is the location of the i^{th} slowest inverter in the top RO. Similarly, let $y_{(i)}=y_j$ where j is the location of the i^{th} slowest inverter in the bottom RO. Because we relax the restriction and allow

the top and bottom ROs to have different configurations, we want to determine the values for each x_i and y_i such that

$$\arg\max_{\mathbf{x},\mathbf{y}} \left| \sum_{i=1}^{n} \Delta d_i \right| \tag{2}$$

$$\text{where } \Delta d_i = \alpha_{(i)} \cdot x_{(i)} - \beta_{(n-i+1)} \cdot y_{(n-i+1)} \text{ , and } \left(\sum_{i=1}^{n} x_i \right) = \left(\sum_{i=1}^{n} y_i \right) \text{ is odd}$$

We sort the delay vectors $\boldsymbol{\alpha}=(\alpha_1, \alpha_2,\ldots,\alpha_n)$ and $\boldsymbol{ß}$ in the descending order. We then pair the slowest inverters in the top RO to the fastest inverters in the bottom RO; the second fastest one in the top RO with the second slowest one in the bottom RO, and so on. Similarly to the solution of Case-1, we can define two sums to keep track the total delay discrepancy when the top RO is faster and when the bottom RO is faster, respectively. The first sum (when the top RO is faster) will stop when the k^{th} fastest inverter in the top RO is not faster than the k^{th} slowest inverter in the bottom RO. The second sum (when the bottom RO is faster) can be computed similarly and like in Case-1, we will construct the RO PUF by selecting the one with the larger magnitude.

The solutions to the above inverter selection problems will be an n-bit configuration vector for each RO. Due to the unpredictable behavior of manufacture variation, we conjecture that the optimal configuration will have about n/2 inverters selected in the ROs. That is, the configuration vector will have roughly half 1's and half 0's. The intuition behind this is that when the systematic variation is filtered out, the true random variation will make inverter delay distribution of the top RO and the bottom RO identical.

4.5 Configurable One-RO-One-Bit (OROB) PUF

Consider the Configurable RO PUF architecture shown in Figure 6, the configuration vector can be used to define the secret bits. For example, if the configurable RO has 10 inverters, we can choose 5 fastest ones and 5 slowest ones (we have to choose odd numbers of inverters to make RO oscillate). Their configuration vectors V1 and V2 should be complementary to each other, assuming V1= (1011011000) and V2= (0100100111). In the traditional RO PUF, the bit is generated not only by the comparison of frequencies, but also two rings' physical locations (the upper one and the lower one). In our RO partition approach, two sub-rings are in the same location, so there is no upper one or lower one. Thus we need to use the configuration information. We can define the sub-ring containing the first inverter as the upper ring, and then the other as the lower ring – so, for example, V1 represents the upper ring, and V2 represents the lower ring. The PUF bit can be defined as 1 if the upper ring is faster than the lower ring; otherwise the PUF bit is 0. Even though the configuration vectors have the spatial mapping relation with the PUF bits – just like the physical location of each RO in traditional PUF – if the attackers happen to know the configuration vectors, they can only know which sub-ring is upper one and lower one. Exposure of configuration vector does not weaken the security of PUF. To get the secrete key, they still need to run the PUF.

Compared with our Configurable Two-ROs-One-Bit (TROB) PUF, our One-RO-One-Bit (OROB) approach has several advantages:

1) Save around 50% hardware without losing the PUF's reliability.
2) Because the fast sub-ring and slow sub-ring share the same path within one RO, and the value of the PUF bit will be determined within this RO rather than comparing with another RO which could be far away from this one, the on-chip spatial variation will have less impact and we can expect the PUF bits to be more reliable.
3) Save the time to solve the inverter selection problem. In our TROB approach, to form an RO pair we first need to subtract the delay information belongs to two ROs; then we need to solve the algorithm for maximizing the largest gap. However, in our OROB approach, we only need to sort the delay data of inverters – so the algorithm complexity is much lower than that of TROB.
4) [35] proved the feasibility of using electromagnetic (EM) emanation to attack the chainwise RO PUF. So it becomes a big threat for the TROB approach, which is based on pairwise ROs. However, in our OROB idea, because two sub-rings make up one RO, these two sub-rings are essentially intertwined together. So even though we cannot claim that EM-emanation attack is infeasible in our situation, it is clear to see that attacking our OROB design is more difficult than the TROB design because of the smaller scale.

4.6 Authentication Using Configuration Vector

The configuration vector can also be used for authentication purpose. For example, we can release the configuration vectors as a public ID of the chip and keep the PUF bits as its secret or private ID. When the chip needs to authenticate itself for non-critical applications, it can use the public ID – on the other hand, the secret PUF bits will be reserved for highly critical authentication applications. In this case, the configuration vectors are also needed to be protected, so we will need to store them in a secured memory.

4.7 Anti-cloning Using Bistable Ring

A bistable ring consists of an even number of inverters as opposed to odd number of inverters in ring oscillator. Unlike a ring oscillator, a bistable ring's output doesn't oscillate – instead, it settles down to a logic-1 or logic-0. Due to our configurable architecture, we can select an odd number of inverters to generate a PUF bit and an even number of inverters to generate an anti-cloning check bit. To clone an RO PUF, the attackers can alter the inverters to make one RO faster than another RO to produce the desirable PUF bit. Because the physical attributes of those inverters are now changed, a bistable ring consists these inverters is no long guaranteed to produce the same bit as before the alteration. As the result, we can tell if a RO PUF is a clone or not by detecting the change in the check bits.

5 Experimental Results

5.1 Simulation

The experiments are conducted based the Virginia Tech's public PUF dataset [36], which consists of frequency measurements of ROs from 198 Xilinx Spartan (XC3S500E) FPGA boards. This dataset only has the frequency measurements of ROs and not individual inverters. We treat each RO as an inverter in our experimentation due to the lack of public data on delay at inverter level. Among the 198 boards, there are 194 boards having the measurements at a fixed supply voltage (1.20V) and a fixed temperature (25°C) [36]. We implement our proposed configurable PUF using the dataset of these 194 boards and extract the PUF output to test:

(A) whether the output information is statistically random, which measures PUF's security;

(B) whether the PUF outputs are unique and collision-free, and

(C) whether the best configurations of different ROs are random and distinct.

In addition, five FPGA boards have measurements taken at varying supply voltages and temperatures [36]. The ranges of supply voltages are 0.96V, 1.08V, 1.20V, 1.32V, and 1.44V. The ranges of temperatures are 35°C, 45°C, 55°C, and 65°C. We use the dataset of these 5 boards to test:

(D) the reliability of our configurable PUFs under different operating environments.

5.1.1 Randomness of Configurable PUF's Output

We use the dataset of 194 FPGA boards with measurement at 1.20V and 25°C. We report the results when each RO consists of 5 inverters, that is n=5. The results for other n values are similar as we will show in later section. Based on the input length requirement (for the purpose of ensuring accuracy) of the NIST statistical test [37], we combine the PUF outputs from 2 FPGA boards to generate a 96-bit output. Therefore, the dataset from the 194 FPGAs provide us 97 PUF-response bit-streams, each with 96 bits long. All these bit-streams are then tested by the NIST statistical test suite for their randomness.

As expected, the NIST test fails on the bit-streams generated from the raw data. This is known to be caused by the systematic variation [38]. After we apply the distiller technique in [38] to filter out the system variation, the new bit-streams successfully pass all the NIST randomness tests.

Table 1 and Table 2 give the detailed test results for our Case-1 and Case-2 configurable PUFs, respectively. According to the NIST test, "The minimum pass rate for each statistical test is approximately = 93 for a sample size = 97 binary sequences". We can see that both Case-1 and Case-2 pass the randomness test on "PROPORTION". For the "P-VALUE" test, the passing threshold is 0.0001 and our bit-steams have scored much higher than this threshold value.

Table 1. NIST TEST RESULTS OF CONFIGURABLE PUFS' OUTPUTS FOR CASE-1.

C1	C2	C3	C4	C5	C6	C7	C8	C9	C10	P-VALUE	PROPORTION	STATISTICAL TEST
11	13	7	10	8	20	10	0	13	5	0.000648	96/97	Frequency
9	14	14	12	9	7	11	10	4	7	0.291249	97/97	BlockFrequency
11	7	12	5	11	13	12	15	4	7	0.118427	96/97	CumulativeSums
12	7	12	9	9	8	10	12	5	13	0.613995	96/97	CumulativeSums
6	14	8	10	10	5	11	9	13	11	0.479268	97/97	Runs
8	13	12	12	7	9	8	11	12	5	0.568055	95/97	Serial
13	11	14	7	10	12	6	9	5	10	0.397299	95/97	Serial

Table 2. NIST TEST RESULTS OF CONFIGURABLE PUFS' OUTPUTS FOR CASE-2.

C1	C2	C3	C4	C5	C6	C7	C8	C9	C10	P-VALUE	PROPORTION	STATISTICAL TEST
9	13	10	9	12	11	13	0	15	5	0.018611	96/97	Frequency
12	16	8	12	9	7	12	7	11	3	0.126333	97/97	BlockFrequency
9	11	9	9	10	7	12	19	7	4	0.060239	96/97	CumulativeSums
9	13	8	13	12	3	7	16	6	10	0.074177	96/97	CumulativeSums
9	8	11	7	8	8	19	10	9	8	0.183769	97/97	Runs
15	5	8	14	7	14	8	7	10	9	0.183769	96/97	Serial
9	13	11	8	13	9	11	9	5	9	0.706149	96/97	Serial

5.1.2 Uniqueness of Configurable PUF's Output

Because PUF output information will be used for many security applications such as device authentication and secret key generation, it is vital to show that different chip will have distinct PUF output. To demonstrate that our configurable PUFs meet this requirement, we compare the pair-wise Hamming distance (HD) on all the 97 96-bit PUF-response bit-streams (see previous section). Figure 10 and Figure 11 are the histogram of the inter-chip HD of outputs of the configurable PUFs for Case-1 and Case-2, where we see the perfect bell shape. For Case-1, the mean HD is 48.83% with a standard deviation of 5.09%. For Case-2, these values are 48.74%

and 5.16%, respectively. Both histograms show that the PUF outputs are unique and it is very unlikely for two PUFs to generate the same output.

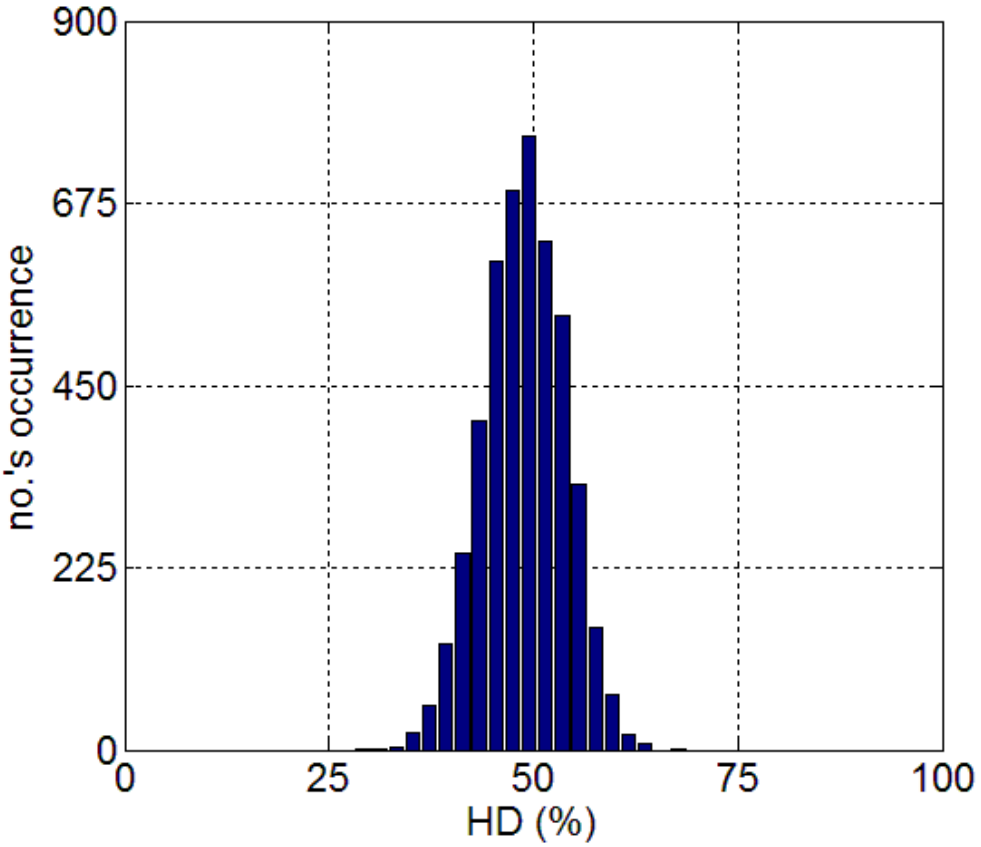

Figure 10. Inter-Chip HD of Configurable PUF Outputs for Case 1.

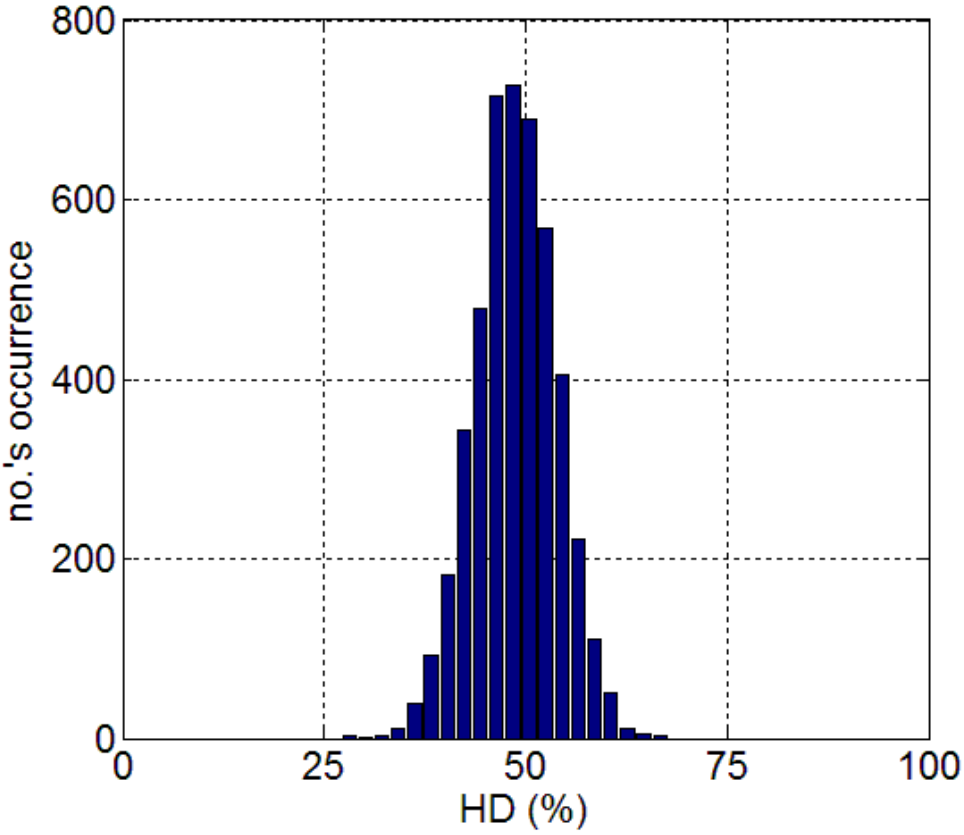

Figure 11. Inter-Chip HD of Configurable PUF Outputs for Case 2.

5.1.3 RO Configuration Information

We study the information of the best configuration of different ROs, that is, which inverters will be selected, because, if these configurations look alike or have certain pattern, it may leak the PUF output information. For each of the 194 FPGA boards, we consider the case that a RO will be constructed from a series of n=15 inverters. Therefore, each FPGA board will have 16 pairs of ROs. For a pair of ROs, we define t_i=1 if the i^{th} inverter in the top RO is included and t_i=0 if it is not selected.

Similarly, b_i=1 means that the i^{th} inverter in the bottom RO is selected. In Case-1, because the top and bottom ROs will have the same configuration, we have t_i=b_i for i=1, 2, …, 15. However, in Case-2, t_i's and b_i's are independent.

In Case-1, the configuration of each RO-pair is defined by a 15-bit bit-stream. Each FPGA board has 16 RO-pairs and we have a total of 194 FPGA boards, this gives us a total of 194*16 = 3104 15-bit bit-streams. We compare their pairwise Hamming distance and find that there is no duplicate (see Figure 12). Indeed, we see that majority of these bit-streams have Hamming distance 6 or 8. In Case-2, because the configuration of the top RO the bottom RO may be different, the configuration of each RO-pair will be characterized by 2 15-bit bit-streams or a 30-bit bit-stream and we will have a total of 194*16 = 3104 30-bit bit-streams. Figure 13 gives the distribution of the pairwise Hamming distance between these bit-streams. We find that there is no duplicate and the majority of these bit-streams have Hamming distance 14 or 16.

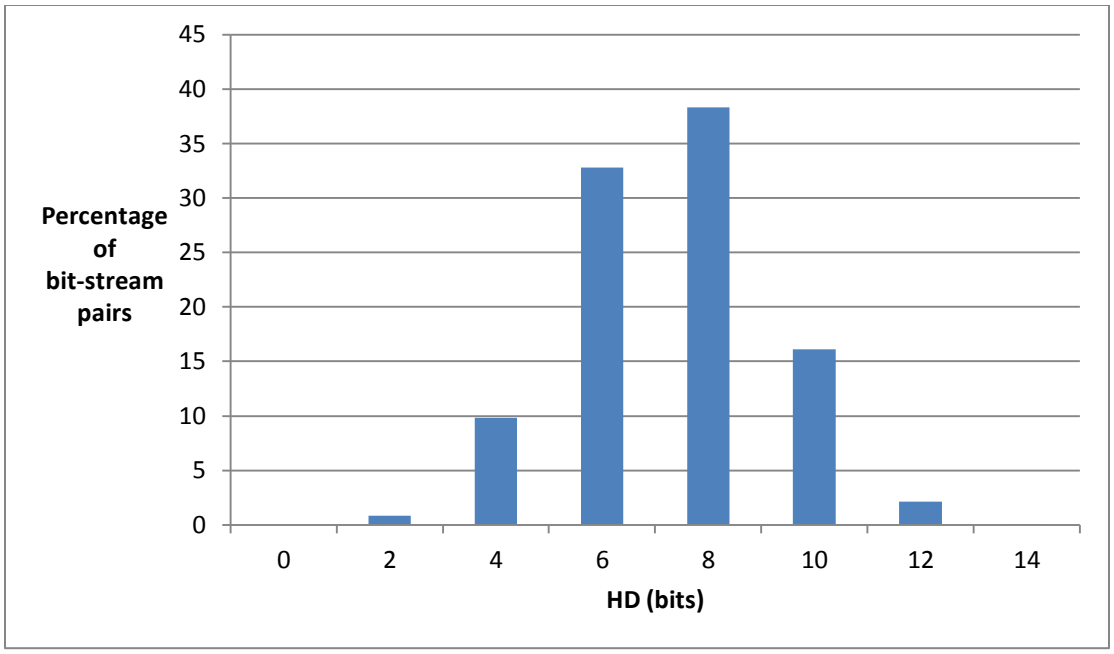

Figure 12. Intra-Chip HD of Best Configuration For Case-1

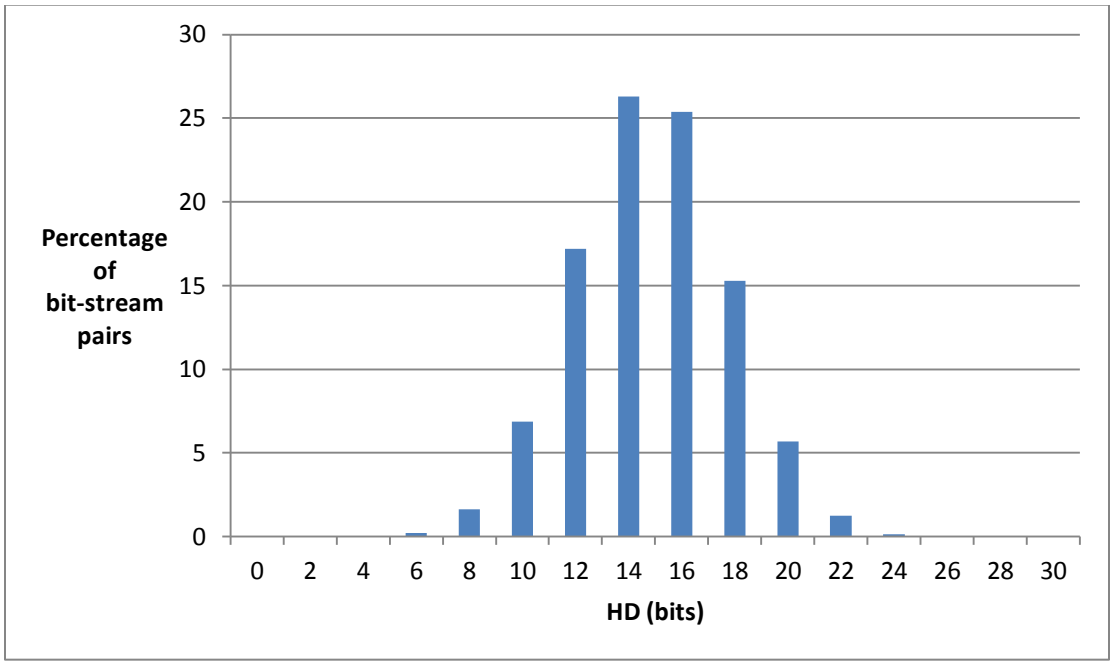

Figure 13. Intra-Chip HD of Best Configuration For Case-2

5.1.4 Reliability

In this part, we demonstrate one of the most important and challenging feature: the reliability of the PUF secret generated from the proposed configurable PUF. To be more specifically,

(1) what will be the best operating environment to test the chip and configure the RO PUF,

(2) whether the bits will flip when the operating environment changes.

With the measurements of five FPGA boards taken under different voltage levels and different temperature, we are able to answer these questions empirically. Let us first consider Case-1 when both top and bottom ROs will have the same configuration.

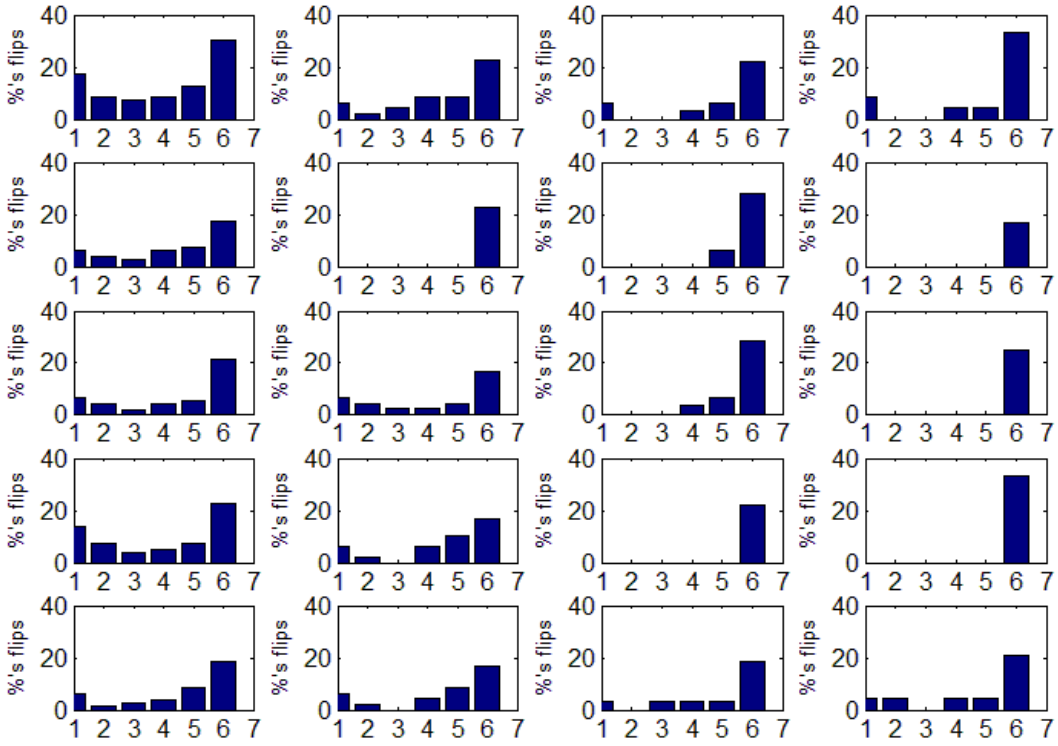

Figure 14. Percentage of Bit Flips under Voltage Variation for Case 1

Figure 14 shows the percentage of bit flips under voltage variation for the proposed configurable RO PUFs, the traditional RO PUFs, and the 1-out-of-8 RO PUFs. It consists of 20 subplots (5 rows and 4 columns). Each row represents the data from each of the five FPGA boards. Each column represents the data for the same number of inverters that are available to be included in the ring oscillator (denoted by n). From left to right, we have n=3, 5, 7, and 9. In each subplot, there are 7 vertical bars representing the percentage of bit flips for a particular FPGA board (row) with a particular n value (column). The first five bars are the percentages of bit flips for the configurable PUFs. The last two are those for the traditional PUFs and

the 1-out-of-8 PUFs. Some bars don't show up (e.g. the last one for the 1-out-of-8 PUFs) because the values of those bars are zero.

Now we report how we determine the bit flips. We extract the baseline PUF outputs based on the measurement at a fixed voltage and fixed temperature (1.20V and 25°C in this case) for the traditional and 1-out-of-8 RO PUFs. The total number of bits in the output is reported in Table 3. Then we extract the PUF outputs from measurements at different voltage levels and check whether there is any difference from the baseline outputs at each bit position. The number of bit positions that have one or multiple changes is considered as the total number of bit flips.

Table 3. TOTAL NUMBER OF BITS PER BOARD

	n=3	**n=5**	**n=7**	**n=9**
Configurable RO PUFs	80	48	32	24
Traditional RO PUFs	80	48	32	24
1-out-of-8 RO PUFs	20	12	8	6

For our configurable RO PUF, because the configuration of the ROs depends on the measurements, we find the best configuration for each possible voltage level, and then test the bit flips with this configuration at other voltage levels. From the first bar to the fifth bar in each subplot, the best configurations are found as voltage goes up from the lowest 0.98V to the highest 1.44V. Note that our method will generate the same number of bits as the traditional RO PUFs.

We make the following observations from Figure 14:

1. The 6th bar, the one for traditional PUF, is always the tallest or the most unreliable. The bars corresponding to the configurable PUFs are much shorter, which means much more reliable than the traditional PUFs.
2. The last bar, the one for 1-out-of-8 PUF, has value 0 in all cases, meaning that 1-out-of-8 is a very reliable PUF. However, it suffers with high hardware cost because it can only generate ¼ of the bits that traditional or our configurable PUFs can generate.
3. The reliability of the configurable PUF increases as we increase n (the length of the ROs) starting from 3. In fact, when n=7, we can achieve the 0% bit flip rate on all five boards.
4. The best configuration determined by using the dataset at the middle voltage value often yields the lowest percentage of bit flips. This suggests the best voltage level to configure the ROs.

Figure 15 shows the impact of temperature variation on the reliability of the PUF outputs. As we can see, only the traditional PUF has bit flips. This suggests that the proposed configurable RO PUFs are very reliable against temperature variations.

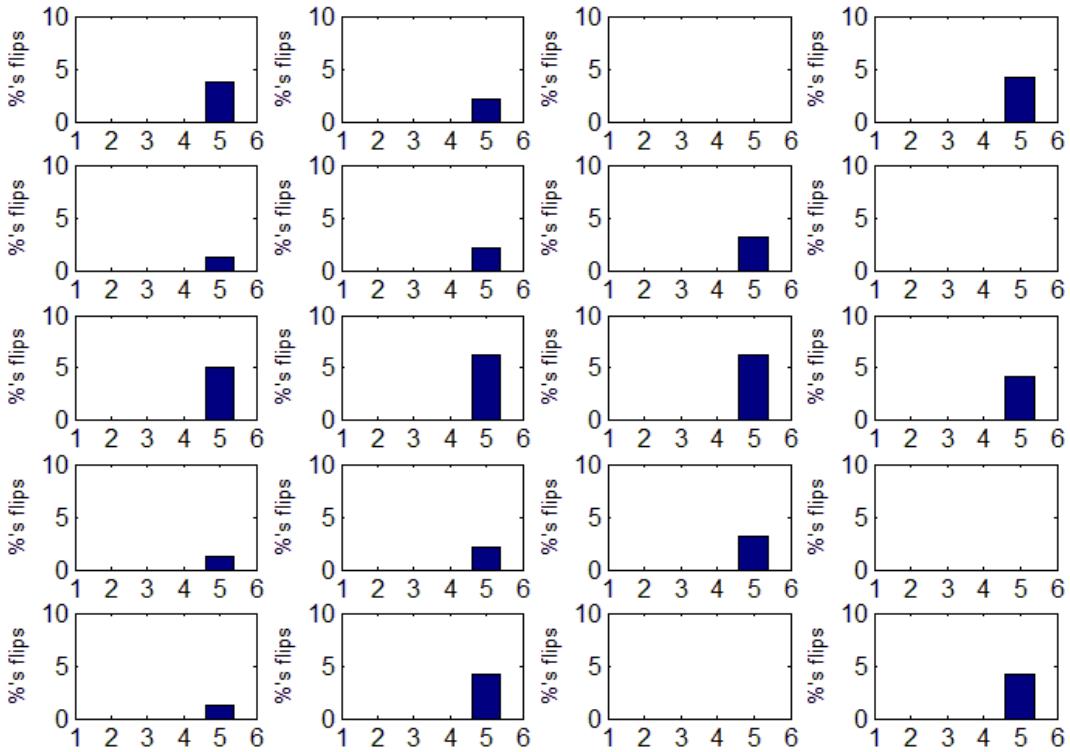

Figure 15. Percentage of Bit Flips under Temperature Variation for Case 1

Figure 16 and Figure 17 shows that similar observations also hold for Case-2 when we allow the top and bottom ROs to be configured differently. The only noticeable difference is that because of this flexibility, the Case-2 configurable PUF becomes more reliable.

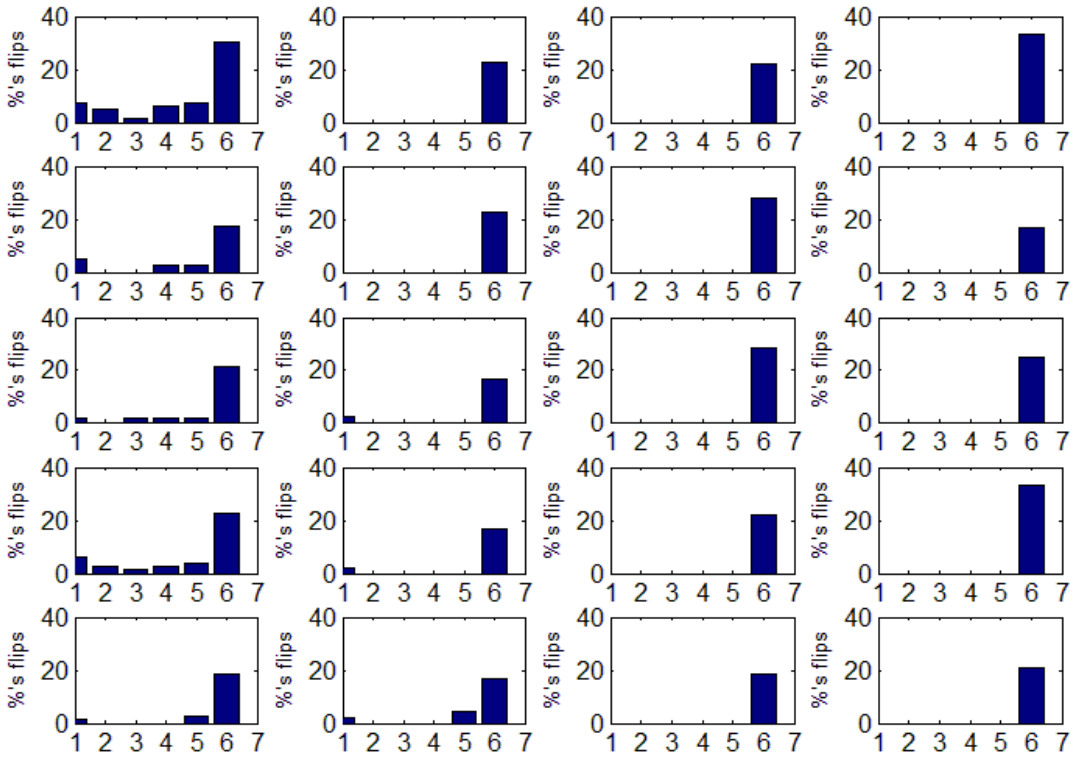

Figure 16. Percentage of Bit Flips under Voltage Variation for Case 2

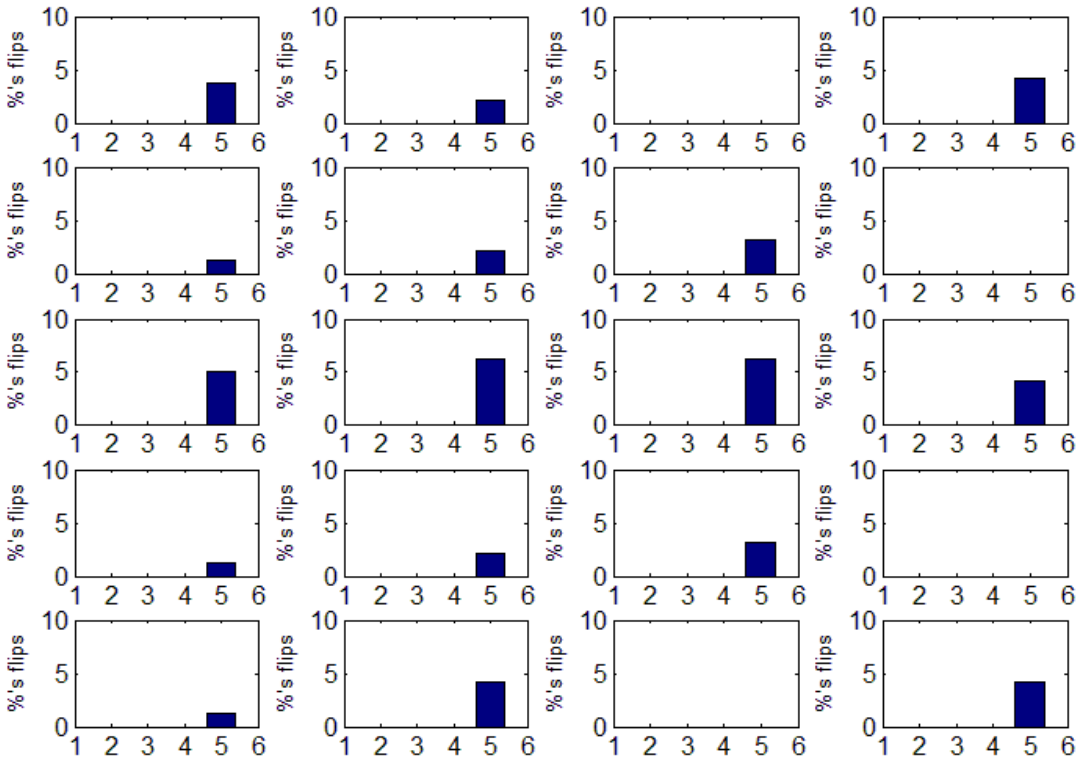

Figure 17. Percentage of Bit Flips under Temperature Variation for Case 2

5.1.5 Uniqueness of OROB Approach

To validate the uniqueness of our OROB approach, we emulate 48 configurable ROs with 10 stages (n=10) on each of 194 Xilinx Spartan (XC3S500E) FPGA boards from VT data. Each configurable RO produces 1 PUF output bit by comparing the fastest and slowest subrings with 5 inverter each (k=5). Figure 18 shows the inter-chip Hamming Distance (HD) of our PUF's output having a normal distribution of mean 48.08% and standard deviation 7.16% and thus it is very unlikely for two PUFs to generate identical outputs.

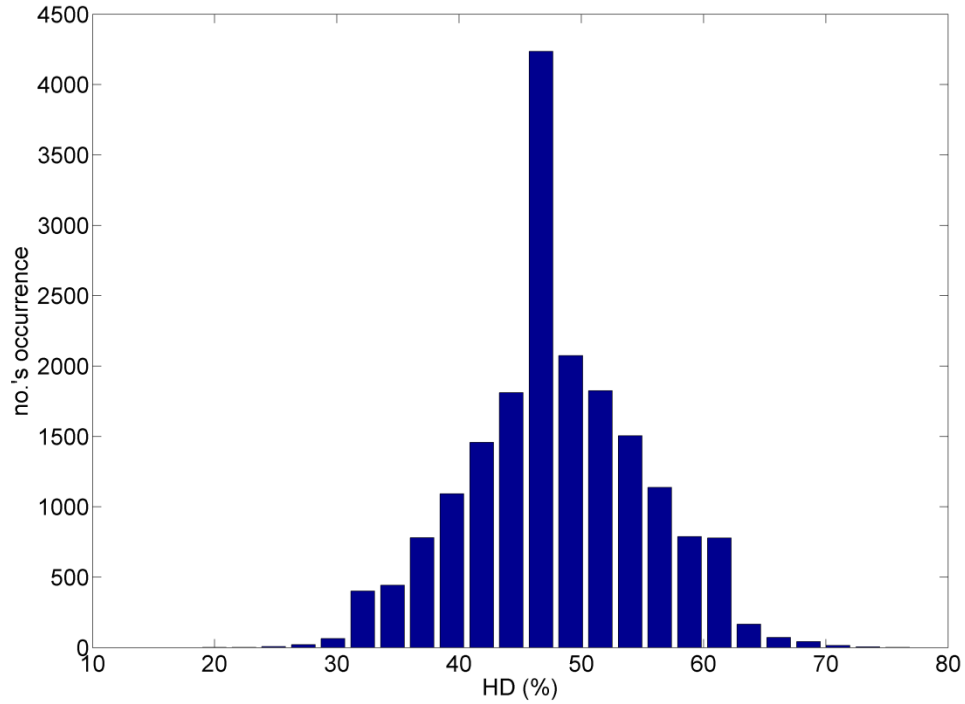

Figure 18. Inter-Chip HD of Configurable OROB PUF Outputs

5.1.6 Reliability of OROB Approach

Five XC3S500E FPGA board from VT data are used to compare the reliability of our OROB approach, traditional RO PUF, and 1-out-of-8 RO PUF under temperature variation (25°C - 65°C). For each board, PUF outputs at room temperature (25°C) are collected and compared to its outputs at the highest operating temperature (65°C) to determine the percentage of bit flips which is shown in Table 5. Clearly the 1-out-of-8 PUF is the most reliable regardless of the RO length and has the highest hardware cost. Our configurable PUF is just as reliable and can generate 4x to 8x as many bits as the 1-out-of-8 PUF. Compared to the traditional PUF, our

configurable PUF is always more reliable and can potentially generate 2x more bits. Among the 3 PUFs, our configurable PUF clearly has the best tradeoff between reliability and hardware cost.

Table 4. PUF BITS PER BOARD (OUR OROB APPROACH VERSUS OTHER RO PUFS)

Trad. 5-stage RO PUF	Trad. 9-stage RO PUF	1/8 5-stage RO PUF	1/8 9-stage RO PUF	Our Config. RO PUF n=10 k=5
48	24	12	6	48

Table 5. RELIABILITY OF OUR OROB APPROACH VERSUS OTHER RO PUFS

Chip Name	Trad. 5-stage RO PUF	Trad. 9-stage RO PUF	1/8 5-stage RO PUF	1/8 9-stage RO PUF	Our Config. RO PUF n=10 k=5
D059546	2.0833	4.1667	0	0	0
D113702	2.0833	0	0	0	0
D113938	6.25	4.1667	0	0	0
D225158	2.0833	0	0	0	0
D225159	4.1667	4.1667	0	0	0
Average	3.3333	2.5	0	0	0

5.1.7 Dual Voltage Scaling Anti-Cloning Scheme

Using the measured delay information of the inverters at 1.20V, we compute the configuration vectors such that some of the secret bits obtained at 0.96V will be flipped at 1.44V; we refer to this as controlled flipping because it is guaranteed that such flipping will always occur. Table 6 shows the controlled flipping patterns for

five XC3S500E FPGA boards from VT data. The second column shows which secret bits will be flipped: a "1" means flipping occurs and a "0" means no flipping occurs. For example, the last hexadecimal digit of Chip3 is E, that means the secret bits at position 1, 2, and 3 are flipped and the secret bit at position 0 is not flipped. The third column shows the percentage of bit flips we can achieve for each chip. Because the flipping rate is high, we can use the flipping pattern to reliably detect a physically cloned PUF. A straightforward way to detect a clone is as follow: (1) set the voltage to 0.96V and get the PUF secret bits R1, (2) set the voltage to 1.44V and get the PUF secret bits R2, and finally (3) because the controlled flipping position F is already pre-computed, we only need to check if F = R1 XOR R2. If yes, the PUF is genuine; otherwise it is a physical clone. This result shows the feasibility of finding the configuration vectors that yields a high flipping rate to detect a physically cloned PUF. Because we use the flipping pattern to detect a physically cloned PUF, we want the pattern to be reliable so we define a threshold R_{th}. If the delay difference between two sub-rings is at least R_{th} at both V1 and V2 and the responses at V1 and V2 are different, we will consider it as a reliable flip and use it for authenticity verification; otherwise we cannot use it.

Table 6. DUAL-VOLTAGE FLIPPING PATTERN

Chip number	Controlled Flipping Position	Flipping rate (%)
Chip1	0x22844202	21.88
Chip2	0xA662BB72	53.13
Chip3	0x2F4480BE	43.75
Chip4	0x12880505	25.00
Chip5	0x3842C280	28.13

5.2 FPGA Implementation

5.2.1 Overview of FPGA Implementation

Arrays of our configurable RO PUF's are implemented on 9 Xilinx Virtex-5 FPGA boards – 3 ML501's, 3 ML506's, and 3 ML510's. The PUFs are operated at 35^{o}C and 70^{o}C so that we can determine the percentage of flips in PUF-response bit-streams under temperature variation. To achieve and maintain the specified temperatures, each FPGA board is placed inside our homemade temperature controlled chamber as shown in Figure 19. A STC-1000 Temperature Controller is used to turn on/off the hair dryer based on the temperature reading from a thermocouple attached to the surface of the Virtex-5 FPGA chip.

Figure 19. Homemade Temperature Controlled Chamber

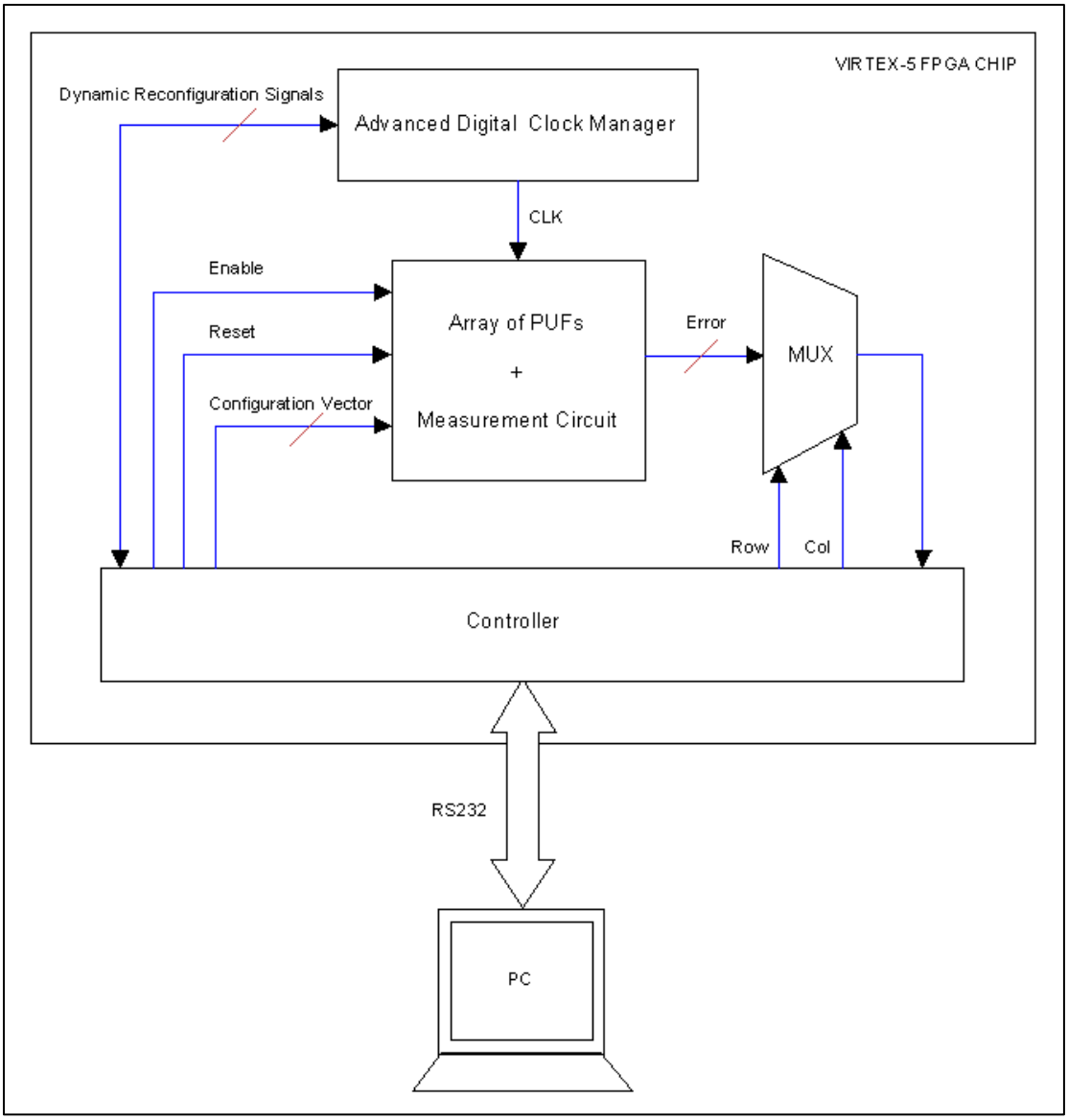

Figure 20. FPGA Implementation of Configurable RO PUFs

Figure 20 provides a diagram of the FPGA implementation of our Configurable RO PUFs. The Advanced Digital Clock Manager (DCM) is a Xilinx primitive that can generate on-the-fly a clock signal with different frequencies and phases without requiring to reprogram the entire FPGA chip – the reconfiguration of the DCM is

done by interacting with its Dynamic Reconfiguration Ports [39]. The clock signal from the DCM is then used to clock the Launch, Sample, and Capture flip-flops of the measurement circuit.

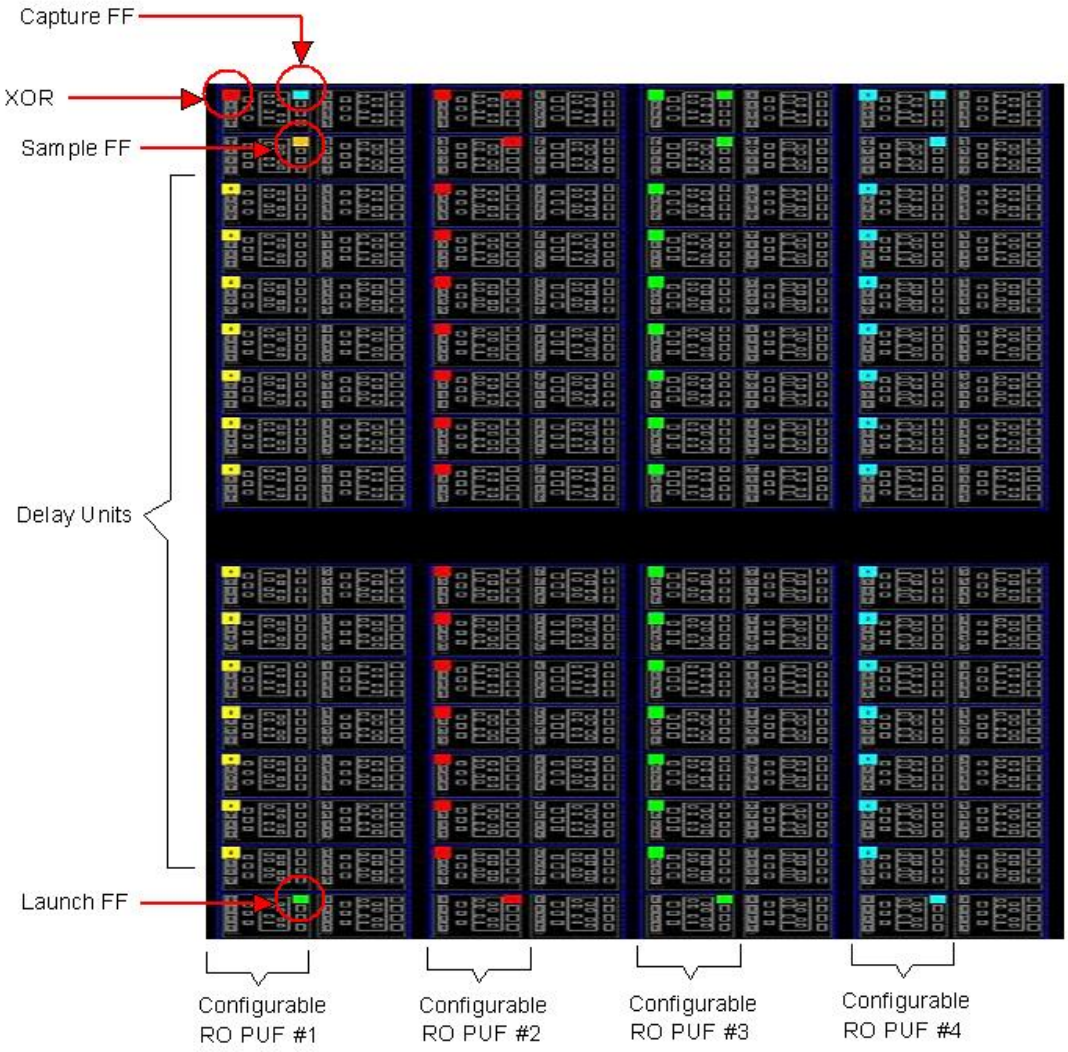

Figure 21. Placement of Configurable RO PUF & Measurement Flip-Flops in Virtex-5

Each Delay Unit in our Configurable RO PUF is implemented by one Look-Up-Table (LUT), which is a specialized circuit that can implement any digital logic circuit and serve as a fundamental building block of Xilinx's FPGA architecture [39]. The locations of LUTs and the measurement flip-flops are manually specified using Relative Location Constraint [40] so that we can achieve identical placement of all Configurable RO PUFs – this helps increasing the uniqueness and randomness of the PUF's outputs. Figure 21 is a snapshot of the actual placement of 4 Configurable RO PUFs and the measurement flip-flops in Xilinx Virtex-5 FPGA chip.

The communication between our FPGA boards and a Windows computer is implemented using RS232 protocol. On the FPGA, we use an open-source transceiver module from Sebastien Bourdeauducq of opencores.org [41] to send/receive information to/from the computer. The software serial port driver is written in C using Window Application Programming Interface (API) [42] and is based on a sample code from knjn.com [43].

5.2.2 Measurement and Analysis Procedure

During the experiment, failure rates of our Configurable RO PUFs at different temperatures are measured on the FPGA boards and then sent to a computer for processing to estimate the delays of Delay Units, calculate the best configuration for each PUF, and determine the reliability and uniqueness of the best-configured PUF-response bit-streams.

In general, the delay of each Delay Unit in a chain consisting of n Delay units can be estimated by a simple algorithm. First, let define 2 functions:

- Measure_All_Zero() : measure the chain delay when setting the configuration vector to all zeroes.
- Meazure(j) : measure the chain delay when setting the configuration vector to all zeros except for j^{th} bit of the vector.

Based on these 2 functions, the algorithm is constructed as the following:

```
1.  d_all_zero = Measure_All_Zero();
2.  for (j=0;  j<(n-1);  i++) {
3.      d[j] = Measure(j);
    }
4.  for (j=0;  j<(n-1);  i++) {
5.      ddiff[j] = d[j] - d_all_zerro;
    }
    // ddiff[j] : the estimated delay of the j^th Delay Unit
```

5.2.3 Uniqueness Result

To validate the uniqueness of our Configurable RO PUF, 9 arrays of Configurable ROs are implemented on the aforementioned 9 FPGA boards, and each Configurable RO consists of 14 Delay Units. The distiller technique in [38] is also applied during post-processing to enhance the uniqueness of the PUF outputs. Figure

22 shows the histogram of the inter-chip HD of the outputs of our Configurable OROB PUFs. This histogram has a bell shape with a mean HD 46.48% and a standard deviation 7.77%. The histogram shows that there is not a pair of FPGA boards that produce identical or completely complementary PUF output stream. Even though the number of FPGA boards used in this experiment is small, this result sill suggests that our Configurable OROB PUF has near-ideal uniqueness.

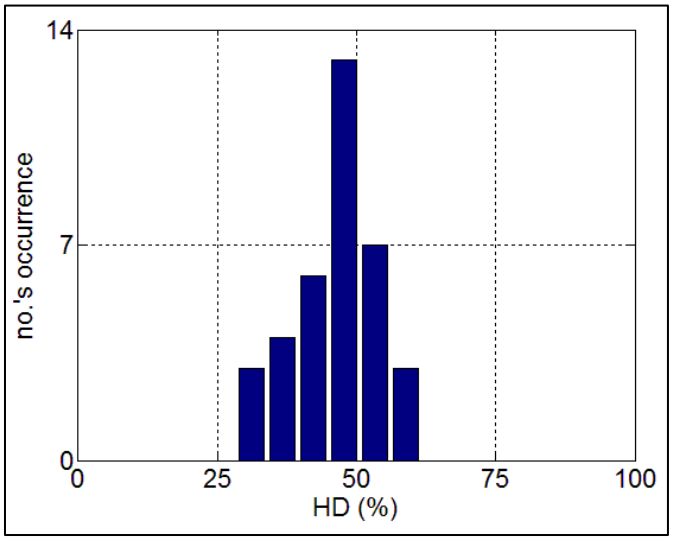

Figure 22. Inter-Chip HD of Configurable PUF Outputs on 9 FPGA boards

5.2.4 Reliability Result

Using the delay information of Delay Units collected from 9 FPGA boards at 35^{o}C and 70^{o}C , we construct our Configurable OROB PUF, Configurable TROB PUF, traditional RO PUF, 1-out-of-8 RO PUF, and neighbor-chained RO PUF with different lengths and compare their PUF outputs' reliability under temperature variation. Table 7 shows the total number of PUF response bits generated from 9

FPGA boards. For our configurable OROB approach, length n means that we compare $\lfloor n/2 \rfloor$ fastest inverters with $\lfloor n/2 \rfloor$ slowest inverters. For other approaches, length n means that every ring oscillator consists of n inverters – we assume n is odd.

Table 7. TOTAL NUMBER OF PUF RESPONSE BITS FROM 9 FPGA BOARDS

Length (n) / *RO PUF Type*	**7**	**11**	**15**
Traditional	1,851	1,134	810
MIT 1-out-of-8	462	279	198
Neighbor-chained	3,693	2,259	1,611
Configurable OROB	3,702	2,268	1,620
Configurable TROB	1,851	1,134	810

Figure 23 shows a quantitative comparison of the reliability of our configurable approaches and other RO PUF designs. The average percentage of bit flips is calculated by averaging the percentages of flips in PUF response bits of 9 FPGA boards under temperature variation. On average, our configurable approaches always generates more reliable PUF response bits than traditional and neighbor-chained approaches regardless of the length of ring oscillators. Our Configurable OROB approach even outperforms the 1-out-of-8 approach and achieves the best reliability.

It is clear from Figure 23 that the 1-out-of-8 approach has the second best reliability but its drawback is its high hardware cost – 4 times as much as the traditional approach. To take both reliability and hardware cost into account, we define a new metric called hardware utilization which is the number of reliable PUF

response bits per one hardware unit. For FPGA implementation, because LUTs are the building block of FPGA architecture, we can consider one LUT as one hardware unit – typically, one inverter is implemented in one LUT in other RO approaches and, in our configurable approaches, we can implement an inverter and a 2-to-1 multiplexer in one LUT. So based on the number of response bits generated by 9 FPGA boards for each approach, we can calculate the corresponding necessary number of LUTs. Then we can calculate the FPGA hardware utilization by dividing the total number of reliable bits generated by 9 FPGA boards by the total number of LUTs. Figure 24 shows our quantitative comparison of the FPGA hardware utilization between our configurable approaches and other RO PUF designs. Clearly, our Configurable OROB approach has the best FPGA hardware utilization regardless of the length of ring oscillators. Even though the neighbor-chained approach has the second best FPGA hardware utilization, its security is much weaker than both of our configurable approaches due to its re-usage of ring oscillators. Therefore, our Configurable OROB approach has the best trade-off among reliability, hardware cost, and security and stands out as the most attractive candidate for FPGA-based RO PUF.

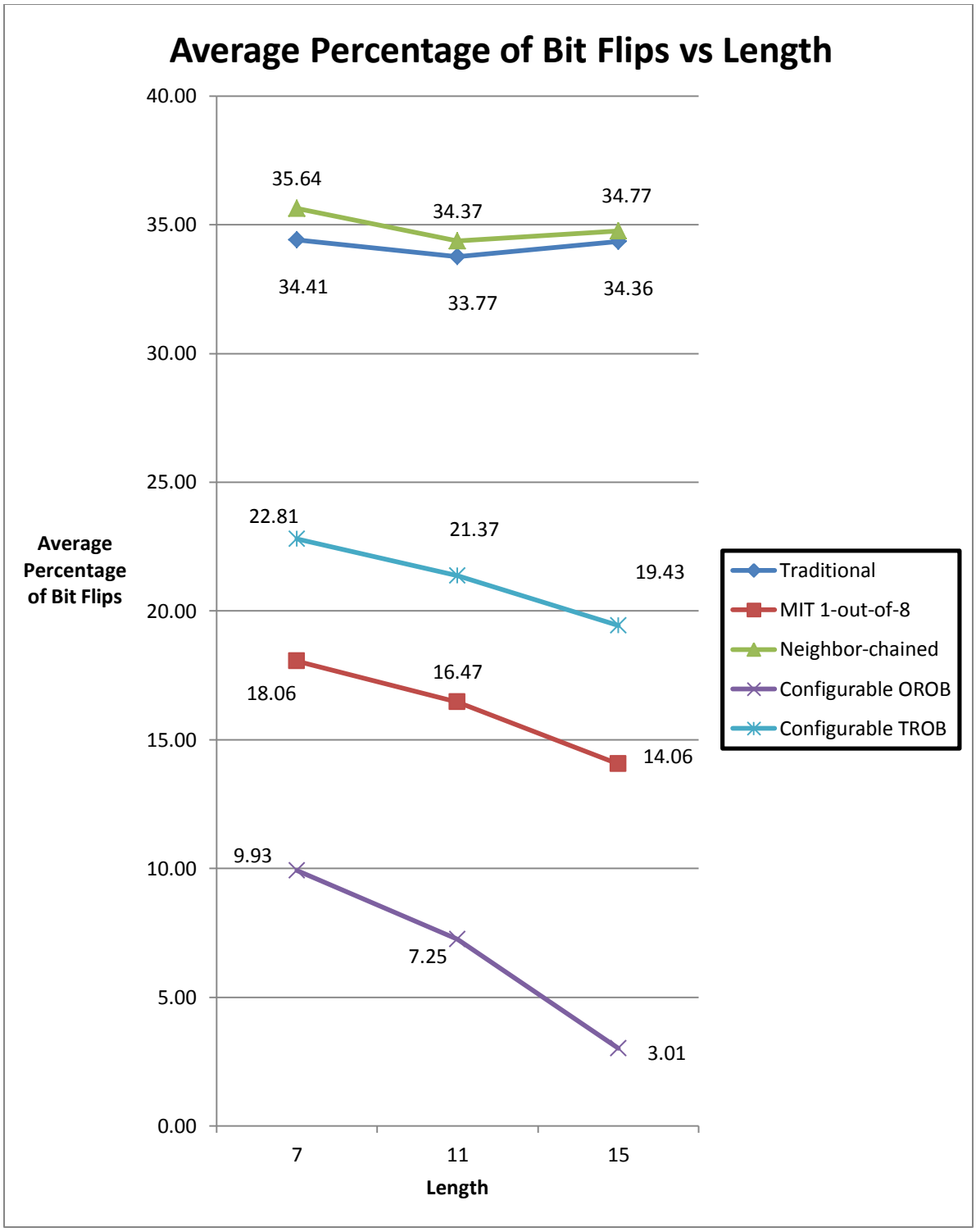

Figure 23. Reliability Comparison of Our Configurable Approach to Other RO PUFs

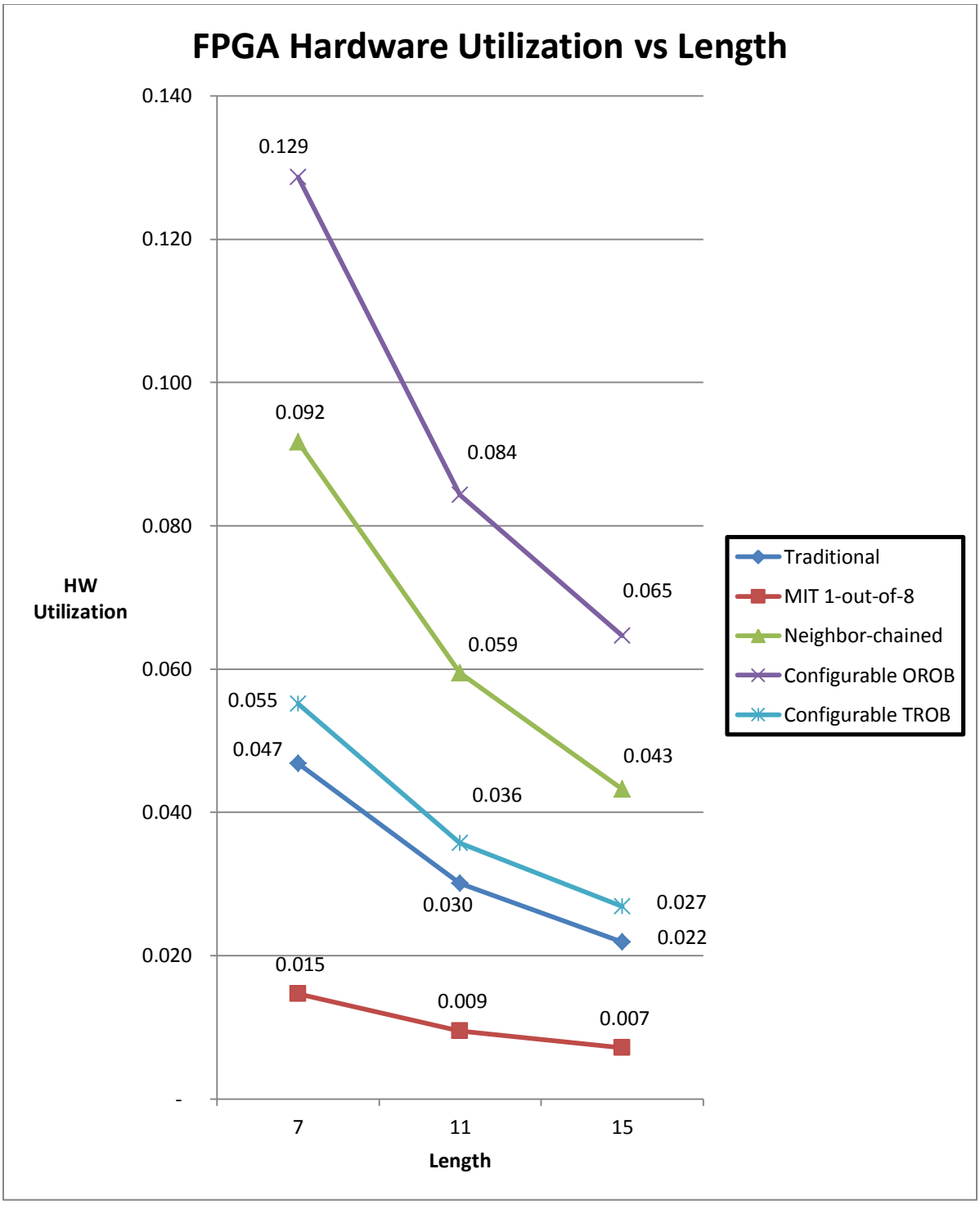

Figure 24. FPGA Hardware Utilization Comparison

6 Conclusion

In this study, we propose a novel configurable RO PUF that can maximize the reliability of the PUF output bits by choosing the best inverters to be included in the ROs after fabrication based on the real inverter delay measurement. Our simulation and FPGA-implementation results show that our configurable approach can outperform the traditional RO PUF and 1-out-of-8 approaches in both reliability and hardware utilization. We also show how the configuration vectors can be used for device authentication, and how the dual-voltage scaling scheme and bistable ring can be used in our approach to detect a cloned RO PUF.

For future study, we can conduct state-of-the-art side-channel attacks and machine-learning-based modelling attacks on our configurable RO PUF. It would be interesting to see if there is any subtle vulnerability in our approach and, if there is, how can we mitigate the vulnerability.

7 Appendices

7.1 Verilog Code

7.1.1 top.v

```
`timescale 1ns / 1ps
`include "global_defs.verh"

module top
# (
  parameter UART_CLK_FREQ = 33000000,
            BAUD = 57600,
            NUM_OF_STAGES_PER_RO = 14,
            NUM_OF_FRD_COLS = 26,
            NUM_OF_FRD_ROWS = 6,
            EHA_SIZE = 32,
            TRANS_COUNTER_SIZE = EHA_SIZE + 8,
            RST_LOGIC_SIZE = 8
)
(
  input CLK_33MHZ_FPGA,
  input FPGA_SERIAL1_RX,
  output FPGA_SERIAL1_TX,
  input rst,
  input loopback_en,
  output [7:0] GPout,
  output GPIO_LED_C,
  output GPIO_LED_E,
  output GPIO_LED_W
);

wire DCM1_CLKIN;
wire DCM1_CLK0;
wire DCM1_CLKFX;
wire DCM1_DRDY;
wire DCM1_LOCKED;
wire DCM1_CLKFB;
wire [6:0] DCM1_DADDR;
wire DCM1_DCLK = DCM1_CLKIN;
wire DCM1_DEN;
wire [15:0] DCM1_DI;
wire DCM1_DWE;
wire DCM1_rst_dcm;

assign GPIO_LED_W = DCM1_LOCKED;
```

```
IBUFG #(
   .IOSTANDARD("DEFAULT")
) DCM1_IBUFG_inst (
   .O(DCM1_CLKIN),
   .I(CLK_33MHZ_FPGA)
);


BUFG DCM1_CLK0_BUFG_inst (
   .O(DCM1_CLKFB),
   .I(DCM1_CLK0)
);


DCM_ADV #(
   .CLKDV_DIVIDE(2.0),
   .CLKFX_DIVIDE(1),
   .CLKFX_MULTIPLY(4),
   .CLKIN_DIVIDE_BY_2("FALSE"),
   .CLKIN_PERIOD(30.30),
   .CLKOUT_PHASE_SHIFT("NONE"),
   .CLK_FEEDBACK("1X"),
   .DCM_PERFORMANCE_MODE("MAX_SPEED"),
   .DESKEW_ADJUST("SYSTEM_SYNCHRONOUS"),
   .DFS_FREQUENCY_MODE("LOW"),
   .DLL_FREQUENCY_MODE("LOW"),
   .DUTY_CYCLE_CORRECTION("TRUE"),
   .FACTORY_JF(16'hf0f0),
   .PHASE_SHIFT(0),
   .SIM_DEVICE("VIRTEX5"),
   .STARTUP_WAIT("FALSE")
) DCM1_ADV_for_puf_inst_inst (
   .CLK0(DCM1_CLK0),
   .CLKFX(DCM1_CLKFX),
   .DRDY(DCM1_DRDY),
   .LOCKED(DCM1_LOCKED),
   .CLKFB(DCM1_CLKFB),
   .CLKIN(DCM1_CLKIN),
   .DADDR(DCM1_DADDR),
   .DCLK(DCM1_DCLK),
   .DEN(DCM1_DEN),
   .DI(DCM1_DI),
   .DWE(DCM1_DWE),
   .PSCLK(1'b0),
   .PSEN(1'b0),
   .PSINCDEC(1'b0),
   .RST(DCM1_rst_dcm)
);
```

```
wire PLL_RST_from_controller;
wire PLL_RST;


BUFG DCM1_CLKFX_BUFG_inst (
   .O(DCM1_CLKFX_BUFG),
   .I(DCM1_CLKFX)
);


arith_shift_right_generic #(RST_LOGIC_SIZE)
DCM1_to_PLL_rst_logc_inst (
  .CLKIN(DCM1_CLKFX_BUFG),
  .RST_IN(PLL_RST_from_controller),
  .I(~DCM1_LOCKED),
  .O(PLL_RST)
);


wire CLKFBOUT;
wire CLKOUT0;
wire PLL_LOCKED;
wire CLKFBDCM;

assign GPIO_LED_C = PLL_LOCKED;


PLL_ADV #(
   .BANDWIDTH("OPTIMIZED"),
   .CLKFBOUT_MULT(16),
   .CLKFBOUT_PHASE(0.0),
   .CLKIN1_PERIOD(30.30),
   .CLKIN2_PERIOD(0.000),
   .CLKOUT0_DIVIDE(16),
   .CLKOUT0_DUTY_CYCLE(0.5),
   .CLKOUT0_PHASE(0.0),
   .CLKOUT1_DIVIDE(1),
   .CLKOUT1_DUTY_CYCLE(0.5),
   .CLKOUT1_PHASE(0.0),
   .CLKOUT2_DIVIDE(1),
   .CLKOUT2_DUTY_CYCLE(0.5),
   .CLKOUT2_PHASE(0.0),
   .CLKOUT3_DIVIDE(1),
   .CLKOUT3_DUTY_CYCLE(0.5),
   .CLKOUT3_PHASE(0.0),
   .CLKOUT4_DIVIDE(1),
   .CLKOUT4_DUTY_CYCLE(0.5),
   .CLKOUT4_PHASE(0.0),
   .CLKOUT5_DIVIDE(1),
   .CLKOUT5_DUTY_CYCLE(0.5),
   .CLKOUT5_PHASE(0.0),
   .COMPENSATION("DCM2PLL"),
```

```
   .DIVCLK_DIVIDE(1),
   .EN_REL("FALSE"),
   .PLL_PMCD_MODE("FALSE"),
   .REF_JITTER(0.100),
   .RST_DEASSERT_CLK("CLKIN1")
) PLL_ADV_inst (
   .CLKFBOUT(CLKFBOUT),
   .CLKOUT0(CLKOUT0),
   .LOCKED(PLL_LOCKED),
   .CLKFBIN(CLKFBOUT),
   .CLKIN1(DCM1_CLKFX_BUFG),
   .CLKINSEL(1'b1),
   .RST(PLL_RST)
);


wire DCM2_RST_from_controller;
wire rst_dcm;
wire CLKOUT0_BUFG;


BUFG CLKOUT0_BUFG_inst (
   .O(CLKOUT0_BUFG),
   .I(CLKOUT0)
);


arith_shift_right_generic #(RST_LOGIC_SIZE)
PLL_to_DCM2_rst_logc_inst (
  .CLKIN(CLKOUT0_BUFG),
  .RST_IN(DCM2_RST_from_controller),
  .I(~PLL_LOCKED),
  .O(rst_dcm)
);


wire UART_CLK = DCM1_CLKIN;
wire CLK0;
wire CLKFX;
wire CLKFX180;
wire [15:0] DO;
wire DRDY;
wire DCM2_LOCKED;
wire CLKFB;
wire [6:0] DADDR;
wire DCLK = UART_CLK;
wire DEN;
wire [15:0] DI;
wire DWE;


assign GPIO_LED_E = DCM2_LOCKED;
```

```
BUFG CLK0_BUFG_inst (
   .O(CLKFB),
   .I(CLK0)
);


DCM_ADV #(
   .CLKDV_DIVIDE(2.0),
   .CLKFX_DIVIDE(1),
   .CLKFX_MULTIPLY(4),
   .CLKIN_DIVIDE_BY_2("FALSE"),
   .CLKIN_PERIOD(30.30),
   .CLKOUT_PHASE_SHIFT("NONE"),
   .CLK_FEEDBACK("1X"),
   .DCM_PERFORMANCE_MODE("MAX_SPEED"),
   .DESKEW_ADJUST("SYSTEM_SYNCHRONOUS"),
   .DFS_FREQUENCY_MODE("LOW"),
   .DLL_FREQUENCY_MODE("LOW"),
   .DUTY_CYCLE_CORRECTION("TRUE"),
   .FACTORY_JF(16'hf0f0),
   .PHASE_SHIFT(0),
   .SIM_DEVICE("VIRTEX5"),
   .STARTUP_WAIT("FALSE")
) DCM_ADV_for_puf_inst_inst (
   .CLK0(CLK0),
   .CLKFX(CLKFX),
   .CLKFX180(CLKFX180),
   .DO(DO),
   .DRDY(DRDY),
   .LOCKED(DCM2_LOCKED),
   .CLKFB(CLKFB),
   .CLKIN(CLKOUT0_BUFG),
   .DADDR(DADDR),
   .DCLK(DCLK),
   .DEN(DEN),
   .DI(DI),
   .DWE(DWE),
   .PSCLK(1'b0),
   .PSEN(1'b0),
   .PSINCDEC(1'b0),
   .RST(rst_dcm)
);


localparam COL_ADDR_WIDTH = `CLOG2(NUM_OF_FRD_COLS);
localparam ROW_ADDR_WIDTH = `CLOG2(NUM_OF_FRD_ROWS);

wire clear_eha;
wire [EHA_SIZE-1 :0] C_late;
wire [TRANS_COUNTER_SIZE-1 :0] C_trans;
```

```
// (* S = "TRUE" *)
wire en_out;
// (* S = "TRUE" *)
wire [COL_ADDR_WIDTH-1 : 0] col_addr;
// (* S = "TRUE" *)
wire [ROW_ADDR_WIDTH-1 : 0] row_addr;
// (* S = "TRUE" *)
wire [NUM_OF_STAGES_PER_RO-1: 0] lut_sel;


// (* KEEP_HIERARCHY = "TRUE" *)
controller
#(
UART_CLK_FREQ,
BAUD,
NUM_OF_STAGES_PER_RO,
NUM_OF_FRD_COLS,
NUM_OF_FRD_ROWS,
EHA_SIZE,
TRANS_COUNTER_SIZE
)
controller_inst (
    .UART_CLK(UART_CLK),
    .FPGA_SERIAL1_RX(FPGA_SERIAL1_RX),
    .FPGA_SERIAL1_TX(FPGA_SERIAL1_TX),
    .rst(rst),
    .loopback_en(loopback_en),
    .C_late(C_late),
    .C_trans(C_trans),
    .en_out(en_out),
    .col_addr(col_addr),
    .row_addr(row_addr),
    .lut_sel(lut_sel),
    .LED(GPout),
    .clear_eha(clear_eha),
    .rst_dcm(DCM2_RST_from_controller),
    .DEN(DEN),
    .DWE(DWE),
    .DADDR(DADDR),
    .DI(DI),
    .DO(DO),
    .DRDY(DRDY),
    .LOCKED(DCM2_LOCKED),
    .DCM1_rst_dcm(DCM1_rst_dcm),
    .DCM1_DEN(DCM1_DEN),
    .DCM1_DWE(DCM1_DWE),
    .DCM1_DADDR(DCM1_DADDR),
    .DCM1_DI(DCM1_DI),
    .DCM1_DRDY(DCM1_DRDY),
    .DCM1_LOCKED(DCM1_LOCKED),
```

```
    .PLL_RST(PLL_RST_from_controller)
);


//(* RLOC_ORIGIN = "X24Y40" *)
puf
#(
NUM_OF_STAGES_PER_RO,
NUM_OF_FRD_COLS,
NUM_OF_FRD_ROWS,
EHA_SIZE
)
puf_inst (
    .en(en_out),
    .CLK(CLKFX),
    .CLK180(CLKFX180),
    .col_addr(col_addr),
    .row_addr(row_addr),
    .lut_sel(lut_sel),
    .clear_eha(clear_eha),
    .C_late(C_late)
);


ripple_counter_generic #(TRANS_COUNTER_SIZE)
trans_counter_inst (
    .value_o(C_trans),
    .ring_osc_i(CLKFX),
    .reset_i(clear_eha),
    .CE(en_out)
);

endmodule
```

7.1.2 arith_shift_right_generic.v

```
`timescale 1ns / 1ps

module arith_shift_right_generic #(parameter SIZE = 8)
(
  input CLKIN,
  input RST_IN,
  input I,
  output O
);

reg [SIZE-1 : 0] shift_reg;

assign O = shift_reg[0];
```

```
always @ (posedge CLKIN or posedge RST_IN)
begin
  if (RST_IN)
    begin
      shift_reg <= {SIZE{1'b1}};
    end
  else
    begin
      shift_reg <= {I,shift_reg[SIZE-1 : 1]};
    end
end

endmodule
```

7.1.3 controller.v

```
`timescale 1ns / 1ps
`include "global_defs.verh"

module controller
# (
  parameter UART_CLK_FREQ = 33000000,
            BAUD = 9600,
            NUM_OF_STAGES_PER_RO = 2,
            NUM_OF_FRD_COLS = 2,
            NUM_OF_FRD_ROWS = 2,
            EHA_SIZE = 32,
            TRANS_COUNTER_SIZE = EHA_SIZE + 8
)
(
  UART_CLK,
  FPGA_SERIAL1_RX,
  FPGA_SERIAL1_TX,
  rst,
  loopback_en,
  C_late,
  C_trans,
  en_out,
  col_addr,
  row_addr,
  lut_sel,
  LED,
  clear_eha,
  rst_dcm,
  DEN,
  DWE,
  DADDR,
  DI,
```

```
  DO,
  DRDY,
  LOCKED,
  DCM1_rst_dcm,
  DCM1_DEN,
  DCM1_DWE,
  DCM1_DADDR,
  DCM1_DI,
  DCM1_DRDY,
  DCM1_LOCKED,
  PLL_RST
);

localparam COL_ADDR_WIDTH = `CLOG2(NUM_OF_FRD_COLS);
localparam ROW_ADDR_WIDTH = `CLOG2(NUM_OF_FRD_ROWS);
localparam NUM_OF_BYTES_FOR_EHA_SIZE = EHA_SIZE/8;
localparam NUM_OF_BYTES_FOR_TRANS_COUNTER_SIZE = TRANS_COUNTER_SIZE/8;
localparam REPORT_EHA_CNT_WIDTH = `CLOG2(NUM_OF_BYTES_FOR_EHA_SIZE);
localparam REPORT_C_TRANS_CNT_WIDTH =
`CLOG2(NUM_OF_BYTES_FOR_TRANS_COUNTER_SIZE);


input UART_CLK;

input FPGA_SERIAL1_RX;
output FPGA_SERIAL1_TX;

input rst;
input loopback_en;

input [EHA_SIZE-1 :0] C_late;
input [TRANS_COUNTER_SIZE-1 :0] C_trans;
output reg en_out;
output reg [COL_ADDR_WIDTH-1 : 0] col_addr;
output reg [ROW_ADDR_WIDTH-1 : 0] row_addr;
output reg [NUM_OF_STAGES_PER_RO-1: 0] lut_sel;
output reg [7:0] LED;
output reg clear_eha;

output reg rst_dcm;
output reg DEN;
output reg DWE;
output reg [6:0] DADDR;
output reg [15:0] DI;
input [15:0] DO;
input DRDY;
input LOCKED;

output reg DCM1_rst_dcm;
output reg DCM1_DEN;
output reg DCM1_DWE;
output reg [6:0] DCM1_DADDR;
```

```
output reg [15:0] DCM1_DI;
input DCM1_DRDY;
input DCM1_LOCKED;
output reg PLL_RST;


wire CLK0_OUT = UART_CLK;
wire dcm_locked =  ~rst;


wire        [15: 0] divisor;
wire        [ 7: 0] rx_data;
wire                rx_irq;
reg         [ 7: 0] tx_data;
reg                 tx_wr;
wire                tx_irq;
wire                tx_tcvr;
wire                arst_n = dcm_locked;

assign divisor = UART_CLK_FREQ/BAUD/16;
assign FPGA_SERIAL1_TX = tx_tcvr;


uart_transceiver transceiver(
      .sys_clk          ( CLK0_OUT ),
      .sys_rst          ( ~arst_n   ),
      .uart_rx          ( FPGA_SERIAL1_RX   ),
      .uart_tx          ( tx_tcvr   ),
      .divisor          ( divisor   ),
      .rx_data          ( rx_data   ),
      .rx_done          ( rx_irq    ),
      .tx_data          ( tx_data   ),
      .tx_wr            ( tx_wr     ),
      .tx_done          ( tx_irq    )
);


`define FSM_STATE_WIDTH 6

localparam RECEIVE                      = `FSM_STATE_WIDTH'd0;
localparam DECODE                       = `FSM_STATE_WIDTH'd1;
localparam STOP_FRD_STATE               = `FSM_STATE_WIDTH'd2;
localparam REPORT_EHA                   = `FSM_STATE_WIDTH'd3;
localparam TX_START                     = `FSM_STATE_WIDTH'd4;
localparam TRANSMIT                     = `FSM_STATE_WIDTH'd5;
localparam DONE                         = `FSM_STATE_WIDTH'd6;

localparam LD_M                         = `FSM_STATE_WIDTH'd7;
localparam LD_D                         = `FSM_STATE_WIDTH'd8;
localparam LD_M_D_W_START               = `FSM_STATE_WIDTH'd9;
localparam LD_M_D_W_WAIT                = `FSM_STATE_WIDTH'd10;
localparam LD_M_D_R_ADDR0               = `FSM_STATE_WIDTH'd11;
```

```
localparam LD_M_D_R_WAIT                    = `FSM_STATE_WIDTH'd12;
localparam LD_M_D_DONE                      = `FSM_STATE_WIDTH'd13;

localparam DFS_FREQ_MODE_R41h_START         = `FSM_STATE_WIDTH'd14;
localparam DFS_FREQ_MODE_R41h_WAIT          = `FSM_STATE_WIDTH'd15;
localparam DFS_FREQ_MODE_R1_ADDR0           = `FSM_STATE_WIDTH'd16;
localparam DFS_FREQ_MODE_R1_ADDR0_WAIT      = `FSM_STATE_WIDTH'd17;
localparam DFS_FREQ_MODE_W41h_START         = `FSM_STATE_WIDTH'd18;
localparam DFS_FREQ_MODE_W41h_WAIT          = `FSM_STATE_WIDTH'd19;
localparam DFS_FREQ_MODE_R2_ADDR0           = `FSM_STATE_WIDTH'd20;
localparam DFS_FREQ_MODE_R2_ADDR0_WAIT      = `FSM_STATE_WIDTH'd21;
localparam DFS_FREQ_MODE_DONE               = `FSM_STATE_WIDTH'd22;

localparam LD_M_DCM1                        = `FSM_STATE_WIDTH'd23;
localparam LD_D_DCM1                        = `FSM_STATE_WIDTH'd24;
localparam LD_M_D_W_START_DCM1              = `FSM_STATE_WIDTH'd25;
localparam LD_M_D_W_WAIT_DCM1               = `FSM_STATE_WIDTH'd26;
localparam LD_M_D_R_ADDR0_DCM1              = `FSM_STATE_WIDTH'd27;
localparam LD_M_D_R_WAIT_DCM1               = `FSM_STATE_WIDTH'd28;
localparam LD_M_D_DONE_DCM1                 = `FSM_STATE_WIDTH'd29;

localparam LD_FRD_COL_ADDR                  = `FSM_STATE_WIDTH'd30;

localparam REPORT_C_TRANS                   = `FSM_STATE_WIDTH'd31;

localparam REPORT_EHA_CONTINUOUSLY          = `FSM_STATE_WIDTH'd32;
localparam UPDATE_TX_DATA                   = `FSM_STATE_WIDTH'd33;


reg [`FSM_STATE_WIDTH-1 : 0] ctr_state;
reg [2:0] op;
reg [4:0] data;
reg [5:0] stop_cnt;
reg [REPORT_EHA_CNT_WIDTH : 0] report_eha_cnt;
reg [15:0] DO_buf;
reg [REPORT_C_TRANS_CNT_WIDTH : 0] report_C_trans_cnt;

always @ ( posedge CLK0_OUT or negedge arst_n)
begin : trans_controller
  if (~arst_n)
  begin
    tx_data           <=  8'd0;
    tx_wr             <=  1'b0;
    ctr_state         <=  RECEIVE;
    op                <=  `NOP;
    data              <=  `NODATA;
    stop_cnt          <=  0;
    report_eha_cnt    <=  0;
    report_C_trans_cnt <= 0;
    en_out            <=  0;
    LED               <=  0;
    clear_eha         <=  1;
```

```
    rst_dcm           <=  0;
    DEN               <=  0;
    DWE               <=  0;
    DADDR             <=  7'hff;
    DO_buf            <=  16'hffff;
    DCM1_rst_dcm      <=  0;
    DCM1_DEN          <=  0;
    DCM1_DWE          <=  0;
    DCM1_DADDR        <=  7'hff;
    PLL_RST           <=  0;
  end
    else
    begin
      if ( loopback_en )
      begin
        tx_data         <=  rx_data;
        tx_wr           <=  rx_irq;
        LED             <=  (rx_irq) ? rx_data : LED;
      end
        else
        begin
          case (ctr_state)
            RECEIVE:
              begin
                if (rx_irq)
                begin
                  op <= rx_data[7:5];
                  data <= rx_data[4:0];
                  ctr_state <= DECODE;
                end
                  else
                  ctr_state <= RECEIVE;
              end

            DECODE:
              begin
                case(op)
                  `LD_COL_ADDR: begin ctr_state <= LD_FRD_COL_ADDR; end
                  `LD_ROW_ADDR: begin row_addr <= data; ctr_state <= DONE;
end
                  `SET_CFG: begin lut_sel[data] <= 1'b1; ctr_state <= DONE;
end
                  `EXT_OP1:
                      begin
                        case (data)
                          `EXT_OP1_CLEAR_CFG: begin lut_sel <= 0; ctr_state
<= DONE; end
                          `EXT_OP1_START_FRD:
                            begin
                              if (LOCKED)
                                begin
                                  en_out <= 1;
```

```
                                clear_eha <= 0;
                                ctr_state <= DONE;
                              end
                            else
                              ctr_state <= DECODE;
                          end
                        `EXT_OP1_STOP_FRD:
                          begin
                            en_out <= 0;
                            stop_cnt <= 0;
                            ctr_state <= STOP_FRD_STATE;
                          end
                        `EXT_OP1_READ_EHA:
                          begin
                            report_eha_cnt <= 0;
                            ctr_state <= REPORT_EHA;
                          end
                        `EXT_OP1_READ_C_TRANS:
                          begin
                            report_C_trans_cnt <= 0;
                            ctr_state <= REPORT_C_TRANS;
                          end
                        `EXT_OP1_CLEAR_EHA:
                          begin
                            clear_eha <= 1;
                            ctr_state <= DONE;
                          end
                        `EXT_OP1_LD_M_D:
                          begin
                            rst_dcm <= 1'b1;
                            DADDR <= 7'h50;
                            ctr_state <= LD_M;
                          end
                        `EXT_OP1_DFS_FREQ_MODE_LOW,
`EXT_OP1_DFS_FREQ_MODE_HIGH:
                          begin
                            rst_dcm <= 1'b1;
                            DADDR <= 7'h41;
                            ctr_state <= DFS_FREQ_MODE_R41h_START;
                          end
                        `EXT_OP1_LD_M_D_DCM1:
                          begin
                            DCM1_rst_dcm <= 1'b1;
                            DCM1_DADDR <= 7'h50;
                            rst_dcm <= 1'b1;
                            PLL_RST <= 1'b1;
                            ctr_state <= LD_M_DCM1;
                          end
                        `EXT_OP1_READ_EHA_CONTINUOUSLY:
                          begin
                            report_eha_cnt <= 0;
                            row_addr <= 0;
```

```
                        col_addr <= 0;
                        ctr_state <= REPORT_EHA_CONTINUOUSLY;
                      end

                    default: ctr_state <= RECEIVE;
                  endcase
                end
            default: ctr_state <= RECEIVE;
          endcase
        end

      DONE: begin tx_data <= {op,data}; ctr_state <= TX_START; end

      STOP_FRD_STATE:
        begin
          stop_cnt <= stop_cnt + 1;
          if (stop_cnt == 60)
            ctr_state <= DONE;
          else
            ctr_state <= STOP_FRD_STATE;
        end

      REPORT_EHA:
        begin
          ctr_state <= TX_START;
          report_eha_cnt <= report_eha_cnt + 1;
          case (report_eha_cnt)
            4'd0: tx_data <= C_late[31:24];
            4'd1: tx_data <= C_late[23:16];
            4'd2: tx_data <= C_late[15:8];
            4'd3: tx_data <= C_late[7:0];
            4'd4: begin data <= `NODATA; ctr_state <= DONE; end
            default: begin data <= `NODATA; ctr_state <= DONE; end
          endcase
        end

      REPORT_C_TRANS:
        begin
          ctr_state <= TX_START;
          report_C_trans_cnt <= report_C_trans_cnt + 1;
          case (report_C_trans_cnt)
            4'd0: tx_data <= C_trans[39:32];
            4'd1: tx_data <= C_trans[31:24];
            4'd2: tx_data <= C_trans[23:16];
            4'd3: tx_data <= C_trans[15:8];
            4'd4: tx_data <= C_trans[7:0];
            4'd5: begin data <= `NODATA; ctr_state <= DONE; end
            default: begin data <= `NODATA; ctr_state <= DONE; end
          endcase
        end

      TX_START: begin tx_wr <= 1'b1; ctr_state <= TRANSMIT; end
```

```
            TRANSMIT:
              begin
                tx_wr <= 1'b0;
                if (tx_irq)
                begin
                  case (op)

                    `EXT_OP1:
                      begin
                        case (data)
                          `EXT_OP1_READ_EHA: begin ctr_state <= REPORT_EHA;
end
                          `EXT_OP1_READ_C_TRANS: begin ctr_state <=
REPORT_C_TRANS; end
                          `EXT_OP1_READ_EHA_CONTINUOUSLY: begin ctr_state
<= REPORT_EHA_CONTINUOUSLY; end
                          default: begin ctr_state <= RECEIVE; end
                        endcase
                      end

                    default: begin ctr_state <= RECEIVE; end
                  endcase
                end
                  else ctr_state <= TRANSMIT;
              end

            LD_M:
              begin
                if (rx_irq)
                begin
                  DI[15:8] <= rx_data;
                  ctr_state <= LD_D;
                end
                  else
                  ctr_state <= LD_M;
              end

            LD_D:
              begin
                if (rx_irq)
                begin
                  DI[7:0] <= rx_data;
                  ctr_state <= LD_M_D_W_START;
                end
                  else
                  ctr_state <= LD_D;
              end

            LD_M_D_W_START:
              begin
                DEN <= 1'b1;
```

```
            DWE <= 1'b1;
            ctr_state <= LD_M_D_W_WAIT;
          end

        LD_M_D_W_WAIT:
          begin
            DEN <= 1'b0;
            DWE <= 1'b0;
            if (DRDY)
              begin
                DADDR <= 7'h00;
                ctr_state <= LD_M_D_R_ADDR0;
              end
            else
              begin
                ctr_state <= LD_M_D_W_WAIT;
              end
          end

        LD_M_D_R_ADDR0:
          begin
            DEN <= 1'b1;
            DWE <= 1'b0;
            ctr_state <= LD_M_D_R_WAIT;
          end

        LD_M_D_R_WAIT:
          begin
            DEN <= 1'b0;
            if (DRDY)
              ctr_state <= LD_M_D_DONE;
            else
              ctr_state <= LD_M_D_R_WAIT;
          end

        LD_M_D_DONE:
          begin
            rst_dcm <= 1'b0;
            op <= `EXT_OP1;
            data <= `EXT_OP1_LD_M_D;
            ctr_state <= DONE;
          end

        DFS_FREQ_MODE_R41h_START:
           begin
            DEN <= 1'b1;
            DWE <= 1'b0;
            ctr_state <= DFS_FREQ_MODE_R41h_WAIT;
          end

        DFS_FREQ_MODE_R41h_WAIT:
          begin
```

```
                DEN <= 1'b0;
                if (DRDY)
                begin
                  if (data == `EXT_OP1_DFS_FREQ_MODE_LOW)
                    DO_buf <= {DO[15:3],1'b0,DO[1:0]};
                  else
                    DO_buf <= {DO[15:3],1'b1,DO[1:0]};
                  DADDR <= 7'h00;
                  ctr_state <= DFS_FREQ_MODE_R1_ADDR0;
                end
                  else
                  ctr_state <= DFS_FREQ_MODE_R41h_WAIT;
              end

            DFS_FREQ_MODE_R1_ADDR0:
              begin
                DEN <= 1'b1;
                DWE <= 1'b0;
                ctr_state <= DFS_FREQ_MODE_R1_ADDR0_WAIT;
              end

            DFS_FREQ_MODE_R1_ADDR0_WAIT:
              begin
                DEN <= 1'b0;
                if (DRDY)
                  begin
                    DADDR <= 7'h41;
                    DI <= DO_buf;
                    ctr_state <= DFS_FREQ_MODE_W41h_START;
                  end
                else
                  ctr_state <= DFS_FREQ_MODE_R1_ADDR0_WAIT;
              end

            DFS_FREQ_MODE_W41h_START:
              begin
                DEN <= 1'b1;
                DWE <= 1'b1;
                ctr_state <= DFS_FREQ_MODE_W41h_WAIT;
              end

            DFS_FREQ_MODE_W41h_WAIT:
              begin
                DEN <= 1'b0;
                DWE <= 1'b0;
                if (DRDY)
                  begin
                    DADDR <= 7'h00;
                    ctr_state <= DFS_FREQ_MODE_R2_ADDR0;
                  end
                else
                  ctr_state <= DFS_FREQ_MODE_W41h_WAIT;
```

```
            end

        DFS_FREQ_MODE_R2_ADDR0:
            begin
              DEN <= 1'b1;
              DWE <= 1'b0;
              ctr_state <= DFS_FREQ_MODE_R2_ADDR0_WAIT;
            end

        DFS_FREQ_MODE_R2_ADDR0_WAIT:
            begin
              DEN <= 1'b0;
              if (DRDY)
                ctr_state <= DFS_FREQ_MODE_DONE;
              else
                ctr_state <= DFS_FREQ_MODE_R2_ADDR0_WAIT;
            end

        DFS_FREQ_MODE_DONE:
            begin
              rst_dcm <= 1'b0;
              ctr_state <= DONE;
            end

        LD_M_DCM1:
            begin
              if (rx_irq)
              begin
                DCM1_DI[15:8] <= rx_data;
                ctr_state <= LD_D_DCM1;
              end
                else
                ctr_state <= LD_M_DCM1;
            end

        LD_D_DCM1:
            begin
              if (rx_irq)
              begin
                DCM1_DI[7:0] <= rx_data;
                ctr_state <= LD_M_D_W_START_DCM1;
              end
                else
                ctr_state <= LD_D_DCM1;
            end

        LD_M_D_W_START_DCM1:
            begin
              DCM1_DEN <= 1'b1;
              DCM1_DWE <= 1'b1;
              ctr_state <= LD_M_D_W_WAIT_DCM1;
            end
```

```
            LD_M_D_W_WAIT_DCM1:
              begin
                DCM1_DEN <= 1'b0;
                DCM1_DWE <= 1'b0;
                if (DCM1_DRDY)
                  begin
                    DCM1_DADDR <= 7'h00;
                    ctr_state <= LD_M_D_R_ADDR0_DCM1;
                  end
                else
                  begin
                    ctr_state <= LD_M_D_W_WAIT_DCM1;
                  end
              end

            LD_M_D_R_ADDR0_DCM1:
              begin
                DCM1_DEN <= 1'b1;
                DCM1_DWE <= 1'b0;
                ctr_state <= LD_M_D_R_WAIT_DCM1;
              end

            LD_M_D_R_WAIT_DCM1:
              begin
                DCM1_DEN <= 1'b0;
                if (DCM1_DRDY)
                  ctr_state <= LD_M_D_DONE_DCM1;
                else
                  ctr_state <= LD_M_D_R_WAIT_DCM1;
              end

            LD_M_D_DONE_DCM1:
              begin
                DCM1_rst_dcm <= 1'b0;
                PLL_RST <= 1'b0;
                rst_dcm <= 1'b0;
                op <= `EXT_OP1;
                data <= `EXT_OP1_LD_M_D_DCM1;
                ctr_state <= DONE;
              end

            LD_FRD_COL_ADDR:
              begin
                if (rx_irq)
                begin
                  col_addr <= rx_data;
                  ctr_state <= DONE;
                end
                  else
                  ctr_state <= LD_FRD_COL_ADDR;
              end
```

```
            REPORT_EHA_CONTINUOUSLY:
              begin
                if (report_eha_cnt == 4)
                  begin
                    report_eha_cnt <=  0;
                    if ((col_addr == NUM_OF_FRD_COLS-1) && (row_addr ==
NUM_OF_FRD_ROWS-1))
                      begin
                       data <= `NODATA;
                       ctr_state <= DONE;
                      end
                    else
                      begin
                        ctr_state <= UPDATE_TX_DATA;
                        if (col_addr == NUM_OF_FRD_COLS-1)
                          begin
                            col_addr <= 0;
                            row_addr <= row_addr + 1;
                          end
                        else
                          begin
                            col_addr <= col_addr + 1;
                          end
                      end
                  end
                else
                  begin ctr_state <= UPDATE_TX_DATA; end
              end

            UPDATE_TX_DATA:
              begin
                ctr_state <= TX_START;
                report_eha_cnt <= report_eha_cnt + 1;
                case (report_eha_cnt)
                  4'd0: tx_data <= C_late[31:24];
                  4'd1: tx_data <= C_late[23:16];
                  4'd2: tx_data <= C_late[15:8];
                  4'd3: tx_data <= C_late[7:0];
                  default: begin data <= `NODATA; ctr_state <= DONE; end
                endcase
              end

            default: ctr_state <= RECEIVE;
          endcase

        end
    end
end

endmodule
```

7.1.4 uart_transceiver.v

```
/*
 * Milkymist VJ SoC
 * Copyright (C) 2007, 2008, 2009, 2010 Sebastien Bourdeauducq
 * Copyright (C) 2007 Das Labor
 *
 * This program is free software: you can redistribute it and/or modify
 * it under the terms of the GNU General Public License as published by
 * the Free Software Foundation, version 3 of the License.
 *
 * This program is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 * GNU General Public License for more details.
 *
 * You should have received a copy of the GNU General Public License
 * along with this program.  If not, see <http://www.gnu.org/licenses/>.
 */

module uart_transceiver(
	input sys_rst,
	input sys_clk,

	input uart_rx,
	output reg uart_tx,

	input [15:0] divisor,

	output reg [7:0] rx_data,
	output reg rx_done,

	input [7:0] tx_data,
	input tx_wr,
	output reg tx_done
);

//----------------------------------------------------------------
// enable16 generator
//----------------------------------------------------------------
reg [15:0] enable16_counter;

wire enable16;
assign enable16 = (enable16_counter == 16'd0);

always @(posedge sys_clk) begin
	if(sys_rst)
		enable16_counter <= divisor - 16'b1;
	else begin
		enable16_counter <= enable16_counter - 16'd1;
		if(enable16)
```

```
                    enable16_counter <= divisor - 16'b1;
      end
end

//----------------------------------------------------------------
// Synchronize uart_rx
//----------------------------------------------------------------
reg uart_rx1;
reg uart_rx2;

always @(posedge sys_clk) begin
      uart_rx1 <= uart_rx;
      uart_rx2 <= uart_rx1;
end

//----------------------------------------------------------------
// UART RX Logic
//----------------------------------------------------------------
reg rx_busy;
reg [3:0] rx_count16;
reg [3:0] rx_bitcount;
reg [7:0] rx_reg;

always @(posedge sys_clk) begin
      if(sys_rst) begin
            rx_done <= 1'b0;
            rx_busy <= 1'b0;
            rx_count16  <= 4'd0;
            rx_bitcount <= 4'd0;
      end else begin
            rx_done <= 1'b0;

            if(enable16) begin
                  if(~rx_busy) begin // look for start bit
                        if(~uart_rx2) begin // start bit found
                              rx_busy <= 1'b1;
                              rx_count16 <= 4'd7;
                              rx_bitcount <= 4'd0;
                        end
                  end else begin
                        rx_count16 <= rx_count16 + 4'd1;

                        if(rx_count16 == 4'd0) begin // sample
                              rx_bitcount <= rx_bitcount + 4'd1;

                              if(rx_bitcount == 4'd0) begin // verify
startbit
                                    if(uart_rx2)
                                          rx_busy <= 1'b0;
                              end else if(rx_bitcount == 4'd9) begin
                                    rx_busy <= 1'b0;
                                    if(uart_rx2) begin // stop bit ok
```

```
                                                rx_data <= rx_reg;
                                                rx_done <= 1'b1;
                                          end // ignore RX error
                                    end else
                                          rx_reg <= {uart_rx2, rx_reg[7:1]};
                              end
                        end
                  end
            end
end

//---------------------------------------------------------------
// UART TX Logic
//---------------------------------------------------------------
reg tx_busy;
reg [3:0] tx_bitcount;
reg [3:0] tx_count16;
reg [7:0] tx_reg;

always @(posedge sys_clk) begin
      if(sys_rst) begin
            tx_done <= 1'b0;
            tx_busy <= 1'b0;
            uart_tx <= 1'b1;
      end else begin
            tx_done <= 1'b0;
            if(tx_wr) begin
                  tx_reg <= tx_data;
                  tx_bitcount <= 4'd0;
                  tx_count16 <= 4'd1;
                  tx_busy <= 1'b1;
                  uart_tx <= 1'b0;
`ifdef SIMULATION
                  $display("UART: %c", tx_data);
`endif
            end else if(enable16 && tx_busy) begin
                  tx_count16  <= tx_count16 + 4'd1;

                  if(tx_count16 == 4'd0) begin
                        tx_bitcount <= tx_bitcount + 4'd1;

                        if(tx_bitcount == 4'd8) begin
                              uart_tx <= 1'b1;
                        end else if(tx_bitcount == 4'd9) begin
                              uart_tx <= 1'b1;
                              tx_busy <= 1'b0;
                              tx_done <= 1'b1;
                        end else begin
                              uart_tx <= tx_reg[0];
                              tx_reg <= {1'b0, tx_reg[7:1]};
                        end
                  end
```

```
            end
      end
end

endmodule
```

7.1.5 puf.v

```
`timescale 1ns / 1ps
`include "global_defs.verh"

module puf
#(
  parameter NUM_OF_STAGES_PER_RO = 2,
            NUM_OF_FRD_COLS = 2,
            NUM_OF_FRD_ROWS = 2,
            EHA_SIZE = 32
)
(
  en,
  CLK,
  CLK180,
  col_addr,
  row_addr,
  lut_sel,
  clear_eha,
  C_late
);

localparam COL_ADDR_WIDTH = `CLOG2(NUM_OF_FRD_COLS);
localparam ROW_ADDR_WIDTH = `CLOG2(NUM_OF_FRD_ROWS);

input en;
input CLK;
input CLK180;
input [COL_ADDR_WIDTH-1 : 0] col_addr;
input [ROW_ADDR_WIDTH-1 : 0] row_addr;
input [NUM_OF_STAGES_PER_RO-1: 0] lut_sel;
input clear_eha;
output [EHA_SIZE-1 : 0] C_late;

wire [EHA_SIZE-1 :0] rsel_mux_out [NUM_OF_FRD_COLS-1 : 0];
wire [EHA_SIZE-1 :0] rsel_mux_in [NUM_OF_FRD_COLS-1 : 0][NUM_OF_FRD_ROWS-1
: 0];
wire [EHA_SIZE-1 :0] csel_mux_out;

assign C_late = csel_mux_out;

genvar rsel_mux_col_i;
```

```
generate
  for (rsel_mux_col_i=0;
       rsel_mux_col_i<NUM_OF_FRD_COLS;
       rsel_mux_col_i=rsel_mux_col_i+1)
  begin : rsel_mux_col
    assign rsel_mux_out[rsel_mux_col_i] =
rsel_mux_in[rsel_mux_col_i][row_addr];
  end
endgenerate

assign csel_mux_out = rsel_mux_out[col_addr];

localparam DIGITS = "9876543210";
wire [NUM_OF_FRD_ROWS-1 : 0] frd_error [NUM_OF_FRD_COLS-1 : 0];

genvar frd_row_i, frd_col_i;
generate
  for (frd_row_i=0; frd_row_i<NUM_OF_FRD_ROWS; frd_row_i=frd_row_i+1)
  begin : frd_matrix_row
    for (frd_col_i=0; frd_col_i<NUM_OF_FRD_COLS; frd_col_i=frd_col_i+1)
    begin : frd_matrix_col

      (* RLOC = {"X", `VAR_TO_STRING(2*frd_col_i) ,"Y",
`VAR_TO_STRING(frd_row_i*(NUM_OF_STAGES_PER_RO+4))} *)
      frd #(NUM_OF_STAGES_PER_RO)
      frd_inst (
        .CLK(CLK),
        .CLK180(CLK180),
        .lut_sel(lut_sel),
        .error(frd_error[frd_col_i][frd_row_i]),
        .en(en)
      );

      ripple_counter_generic #(EHA_SIZE)
      eha_inst (
          .value_o(rsel_mux_in[frd_col_i][frd_row_i]),
          .ring_osc_i(frd_error[frd_col_i][frd_row_i]),
          .reset_i(clear_eha),
          .CE(en)
      );

    end
  end
endgenerate

endmodule
```

7.1.6 frd.v

```
`timescale 1ns / 1ps
`include "global_defs.verh"

module frd #(parameter NUM_OF_STAGES_PER_RO = 2)
(
  input CLK,
  input CLK180,
  input [NUM_OF_STAGES_PER_RO-1: 0] lut_sel,
  output error,
  input en
);

wire S;
wire EN_LR_SR_EDC = en;
wire D;
wire Q;
wire E = D ^ Q;
wire L;
wire TR_Q;
assign error = TR_Q;


(* RLOC = {"X0Y0"} *)
FDRSE #(
   .INIT(1'b0)
) LR_inst (
   .Q(S),
   .C(CLK),
   .CE(EN_LR_SR_EDC),
   .D(~S),
   .R(1'b0),
   .S(1'b0)
);


(* RLOC = {"X0Y1"} *)
cfg_ro #(NUM_OF_STAGES_PER_RO)
cfg_ro_inst (
  .cfg_ro_in(S),
  .lut_sel(lut_sel),
  .cfg_ro_out(D)
);


localparam DIGITS = "9876543210";
localparam Y_SR = NUM_OF_STAGES_PER_RO + 1;

(* RLOC = {"X0Y", `VAR_TO_STRING(Y_SR)} *)
FDRSE #(
```

```
   .INIT(1'b0)
) SR_inst (
   .Q(Q),
   .C(CLK180),
   .CE(EN_LR_SR_EDC),
   .D(D),
   .R(1'b0),
   .S(1'b0)
);


localparam Y_CR = Y_SR + 1;

(* RLOC = {"X0Y", `VAR_TO_STRING(Y_CR)} *)
FDRSE #(
   .INIT(1'b0)
) CR_inst (
   .Q(L),
   .C(CLK),
   .CE(EN_LR_SR_EDC),
   .D(E),
   .R(1'b0),
   .S(1'b0)
);


localparam Y_TR = Y_CR + 1;

(* RLOC = {"X0Y", `VAR_TO_STRING(Y_TR)} *)
FDRSE #(
   .INIT(1'b0)
) TR_inst (
   .Q(TR_Q),
   .C(CLK),
   .CE(L),
   .D(~TR_Q),
   .R(1'b0),
   .S(1'b0)
);

endmodule
```

7.1.7 cfg_ro.v

```
`timescale 1ns / 1ps
`include "global_defs.verh"

module cfg_ro #(parameter NUM_OF_STAGES_PER_RO = 2)
(
```

```
  input cfg_ro_in,
  input [NUM_OF_STAGES_PER_RO-1: 0] lut_sel,
  output cfg_ro_out
);

wire [NUM_OF_STAGES_PER_RO  : 0] lut_out;

assign lut_out[NUM_OF_STAGES_PER_RO] = cfg_ro_in;
assign cfg_ro_out = lut_out[0];

localparam DIGITS = "9876543210";
genvar i;
generate
  for (i=0; i<NUM_OF_STAGES_PER_RO; i=i+1) begin : stage

    (* RLOC = {"X0Y", `VAR_TO_STRING(i)} *)
    LUT6 #(
       .INIT(64'h0000FFFFFFFF0000)
    ) LUT6_inst (
       .O(lut_out[i]),
       .I4(lut_out[i+1]),
       .I5(lut_sel[i])
    );
  end
endgenerate

endmodule
```

7.1.8 ripple_counter_generic.v

```
`timescale 1ns / 1ps

module ripple_counter_generic #(parameter SIZE = 32)
(
  output [SIZE-1 : 0] value_o,
  input ring_osc_i,
  input reset_i,
  input CE
);

wire [SIZE : 0] toggle_dff_o;
wire [SIZE : 1] CE_dff;

assign value_o = toggle_dff_o[SIZE : 1];
assign toggle_dff_o[0] = ring_osc_i;

// synthesis attribute keep [of] toggle_dff_o [is] "true";
```

```
generate
       genvar i;

       for (i=1; i<=SIZE; i=i+1) begin : toggle_dff

    if (i==1)
      assign CE_dff[i] = CE;
    else
      assign CE_dff[i] = 1'b1;

    FDCPE_1 #(
       .INIT(1'b0)
    ) FDCPE_inst (
       .Q(toggle_dff_o[i]),
       .C(toggle_dff_o[i-1]),
       .CE(CE_dff[i]),
       .CLR(reset_i),
       .D(~toggle_dff_o[i]),
       .PRE(1'b0)
    );


       end
endgenerate

endmodule
```

7.1.9 global_defs.verh

```
`define CLOG2(x) \
   (x ==      0) ?  -1 : \
   (x ==      1) ?   0 : \
   (x <=      2) ?   1 : \
   (x <=      4) ?   2 : \
   (x <=      8) ?   3 : \
   (x <=     16) ?   4 : \
   (x <=     32) ?   5 : \
   (x <=     64) ?   6 : \
   (x <=    128) ?   7 : \
   (x <=    256) ?   8 : \
   (x <=    512) ?   9 : \
   (x <=   1024) ?  10 : \
   (x <=   2048) ?  11 : \
   (x <=   4096) ?  12 : \
   (x <=   8192) ?  13 : \
   (x <=  16384) ?  14 : \
   (x <=  32768) ?  15 : \
   (x <=  65536) ?  16 : \
   (x <= 131072) ?  17 : \
```

```
   (x <= 262144) ?  18 : \
   -1



`define THOUSANDS(x) (x / 1000)
`define  HUNDREDS(x) ((x - (`THOUSANDS(x) * 1000)) / 100)
`define      TENS(x) ((x - (`THOUSANDS(x) * 1000) - (`HUNDREDS(x) * 100)) /
10)
`define      ONES(x) (x - (`THOUSANDS(x) * 1000) - (`HUNDREDS(x) * 100) -
(`TENS(x) * 10))

`define     TO_STRING(x) (DIGITS[((8 * (x + 1)) - 1) : (8 * x)])
`define VAR_TO_STRING(x) ({`TO_STRING(`THOUSANDS(x)),
`TO_STRING(`HUNDREDS(x)), `TO_STRING(`TENS(x)), `TO_STRING(`ONES(x))})


`define NOP                                3'h0
`define LD_COL_ADDR                        3'h1
`define LD_ROW_ADDR                        3'h2
`define SET_CFG                            3'h3
`define EXT_OP1                            3'h7

`define NODATA                             5'h0
`define EXT_OP1_CLEAR_CFG                  5'h1
`define EXT_OP1_START_FRD                  5'h2
`define EXT_OP1_STOP_FRD                   5'h3
`define EXT_OP1_READ_EHA                   5'h4
`define EXT_OP1_CLEAR_EHA                  5'h5
`define EXT_OP1_LD_M_D                     5'h6
`define EXT_OP1_DFS_FREQ_MODE_LOW          5'h7
`define EXT_OP1_DFS_FREQ_MODE_HIGH         5'h8
`define EXT_OP1_LD_M_D_DCM1                5'h9
`define EXT_OP1_READ_C_TRANS               5'ha
`define EXT_OP1_READ_EHA_CONTINUOUSLY      5'hb


`define NULL                           0
```

7.2 C Code

7.2.1 main.cpp

```
#include <windows.h>
#include <stdio.h>
```

```
#include <conio.h>
#include <string.h>
#include <math.h>
#include <time.h>
#include "defs.h"

HANDLE hCom;

void OpenCom()
{
       DCB dcb;
       COMMTIMEOUTS ct;

       hCom = CreateFile( TEXT("COM3"), GENERIC_READ | GENERIC_WRITE, 0,
NULL, OPEN_EXISTING, FILE_ATTRIBUTE_NORMAL, NULL);

       if(hCom==INVALID_HANDLE_VALUE) exit(1);
       if(!SetupComm(hCom, 4096, 4096)) exit(1);

   if(!GetCommState(hCom, &dcb)) exit(1);
       dcb.BaudRate = BAUD_RATE;
       ((DWORD*)(&dcb))[2] = 0x1001;
       dcb.ByteSize = 8;
       dcb.Parity = NOPARITY;
       dcb.StopBits = 2;
       if(!SetCommState(hCom, &dcb)) exit(1);

       ct.ReadIntervalTimeout = MAXDWORD;
       ct.ReadTotalTimeoutMultiplier = MAXDWORD;
       ct.ReadTotalTimeoutConstant = 100;
       ct.WriteTotalTimeoutMultiplier = 0;
       ct.WriteTotalTimeoutConstant = 0;

       if(!SetCommTimeouts(hCom, &ct))
   {
      exit(1);
   }
}

void CloseCom()
{
       CloseHandle(hCom);
}

DWORD WriteCom(unsigned char* buf, int len)
{
       DWORD nSend;
       if(!WriteFile(hCom, buf, len, &nSend, NULL)) exit(1);

       return nSend;
}
```

```
void WriteComChar(unsigned char b)
{
       WriteCom(&b, 1);
}

int ReadCom(unsigned char *buf, int len)
{
       DWORD nRec;
       if(!ReadFile(hCom, buf, len, &nRec, NULL)) exit(1);

       return (int)nRec;
}

unsigned char ReadComChar()
{
       DWORD nRec;
       unsigned char c;
       if(!ReadFile(hCom, &c, 1, &nRec, NULL))
   {
      exit(1);
   }

   if (nRec)
   {
#ifdef DEBUG_PRINT_SUCCEED_READCOMCHAR
      printf("\n %s(), line %d: lpNumberOfBytesRead = %d;  Receive_Char =
0x%X\n", __FUNCTION__, __LINE__, nRec, c);
#endif
      return c;
   }
   else
   {
      printf("\n %s(), line %d: lpNumberOfBytesRead = %d\n", __FUNCTION__,
__LINE__, nRec);
      return 0;
   }
}


void loadcr(loadcr_CHOICE_T choice, unsigned int valin)
{
   if (((choice == COL) && (valin > COL_MAX)) || ((choice == ROW) && (valin
> ROW_MAX))) {
      printf("\nERROR: %s(), line %d: Exit ...\n", __FUNCTION__, __LINE__);
getchar(); exit(1);
   }

   unsigned char ctx, crx;

   switch (choice) {
      case COL:
         ctx = (unsigned char)(LD_COL_ADDR << OP_POS);
```

```
         WriteComChar(ctx);
         ctx = valin;
         WriteComChar(ctx);
         ctx = (unsigned char)(LD_COL_ADDR << OP_POS);
         break;
      case ROW:
         ctx = (unsigned char)(LD_ROW_ADDR << OP_POS);
         ctx = ctx | (unsigned char)(valin);
         WriteComChar(ctx);
         break;
      default:
         printf("\nERROR: %s(), line %d: Exit ...\n", __FUNCTION__,
__LINE__); getchar(); exit(1);
         break;
   }

   crx = ReadComChar();
   if (crx == ctx)
   {
#ifdef DEBUG_PRINT_SUCCEED_CONFIRMATION
      printf("\n%s succeeds: ctx 0x%X, crx 0x%X\n", __FUNCTION__, ctx,
crx);
#endif
      return;
   }
   else
   {
      printf("\n%s fails: ctx 0x%X, crx 0x%X\n", __FUNCTION__, ctx, crx);
      printf("\nERROR: %s(), line %d: Exit ...\n", __FUNCTION__, __LINE__);
getchar(); exit(1);
   }
}


void sccfg(bool choice, unsigned int cfg)
{
   unsigned char ctx, crx;
   if (cfg > CFG_MAX)
   {
      printf("\nsccfg() ERROR : cfg is out of bound\n");
      printf("\nERROR: %s(), line %d: Exit ...\n", __FUNCTION__, __LINE__);
getchar(); exit(1);
   }

   if (choice == CLR)
      ctx = (unsigned char)((EXT_OP1 << OP_POS) | (EXT_OP1_CLEAR_CFG));
   else
      ctx = (unsigned char)((SET_CFG << OP_POS) | cfg);

   WriteComChar(ctx);
   crx = ReadComChar();
   if (crx == ctx)
```

```
   {
#ifdef DEBUG_PRINT_SUCCEED_CONFIRMATION
      printf("\nsccfg() succeeds: ctx 0x%X, crx 0x%X\n", ctx, crx);
#endif
      return;
   }
   else
   {
      printf("\nsccfg() fails: ctx 0x%X, crx 0x%X\n", ctx, crx);
      printf("\nERROR: %s(), line %d: Exit ...\n", __FUNCTION__, __LINE__);
getchar(); exit(1);
   }
}


void read_C_late(unsigned char* buf)
{
   unsigned char ctx = (unsigned char)((EXT_OP1 << OP_POS) |
(EXT_OP1_READ_EHA));
   unsigned char crx;

   WriteComChar(ctx);

   for (int i=0; i<EHA_BYTE_SIZE; i++)
   {
      crx = ReadComChar();
      buf[i] = crx;
   }

   crx = ReadComChar();
   ctx = (unsigned char)((EXT_OP1 << OP_POS) | NODATA);
   if (crx == ctx)
   {
#ifdef DEBUG_PRINT_SUCCEED_CONFIRMATION
      printf("\n%s() succeeds: ctx 0x%X, crx 0x%X\n", __FUNCTION__, ctx,
crx);
#endif
      return;
   }
   else
   {
      printf("\n%s() fails: ctx 0x%X, crx 0x%X\n", __FUNCTION__, ctx, crx);
      printf("\nERROR: %s(), line %d: Exit ...\n", __FUNCTION__, __LINE__);
getchar(); exit(1);
   }
}


void read_C_late_continuously(unsigned char* buf)
{
   unsigned char ctx = (unsigned char)((EXT_OP1 << OP_POS) |
(EXT_OP1_READ_EHA_CONTINUOUSLY));
```

```
   unsigned char crx;

   WriteComChar(ctx);

   for (int i=0; i<EHA_BYTE_SIZE*FRD_TOT; i++)
   {
      crx = ReadComChar();
      buf[i] = crx;
   }

   crx = ReadComChar();
   ctx = (unsigned char)((EXT_OP1 << OP_POS) | NODATA);
   if (crx == ctx)
   {
#ifdef DEBUG_PRINT_SUCCEED_CONFIRMATION
      printf("\n%s() succeeds: ctx 0x%X, crx 0x%X\n", __FUNCTION__, ctx,
crx);
#endif
      return;
   }
   else
   {
      printf("\n%s() fails: ctx 0x%X, crx 0x%X\n", __FUNCTION__, ctx, crx);
      printf("\nERROR: %s(), line %d: Exit ...\n", __FUNCTION__, __LINE__);
getchar(); exit(1);
   }
}


void read_C_trans(unsigned char* buf)
{
   unsigned char ctx = (unsigned char)((EXT_OP1 << OP_POS) |
(EXT_OP1_READ_C_TRANS));
   unsigned char crx;

   WriteComChar(ctx);

   for (int i=0; i<TRANS_COUNTER_BYTE_SIZE; i++)
   {
      crx = ReadComChar();
      buf[i] = crx;
   }

   crx = ReadComChar();
   ctx = (unsigned char)((EXT_OP1 << OP_POS) | NODATA);
   if (crx == ctx)
   {
#ifdef DEBUG_PRINT_SUCCEED_CONFIRMATION
      printf("\n%s() succeeds: ctx 0x%X, crx 0x%X\n", __FUNCTION__, ctx,
crx);
#endif
      return;
```

```
   }
   else
   {
      printf("\n%s() fails: ctx 0x%X, crx 0x%X\n", __FUNCTION__, ctx, crx);
      printf("\nERROR: %s(), line %d: Exit ...\n", __FUNCTION__, __LINE__);
getchar(); exit(1);
   }
}


void clear_eha()
{
   unsigned char ctx = (unsigned char)((EXT_OP1 << OP_POS) |
(EXT_OP1_CLEAR_EHA));
   unsigned char crx;

   WriteComChar(ctx);
   crx = ReadComChar();
   if (crx == ctx)
   {
#ifdef DEBUG_PRINT_SUCCEED_CONFIRMATION
      printf("\nclear_eha() succeeds: ctx 0x%X, crx 0x%X\n", ctx, crx);
#endif
      return;
   }
   else
   {
      printf("\nclear_eha() fails: ctx 0x%X, crx 0x%X\n", ctx, crx);
      printf("\nERROR: %s(), line %d: Exit ...\n", __FUNCTION__, __LINE__);
getchar(); exit(1);
   }
}


void sta_stofrd(bool choice)
{
   unsigned char ctx, crx;

   if (choice == START)
      ctx = (unsigned char)((EXT_OP1 << OP_POS) | (EXT_OP1_START_FRD));
   else
      ctx = (unsigned char)((EXT_OP1 << OP_POS) | (EXT_OP1_STOP_FRD));

   WriteComChar(ctx);
   crx = ReadComChar();
   if (crx == ctx)
   {
#ifdef DEBUG_PRINT_SUCCEED_CONFIRMATION
      printf("\nsta_stofrd() succeeds: ctx 0x%X, crx 0x%X\n", ctx, crx);
#endif
      return;
   }
```

```
   else
   {
      printf("\nsta_stofrd() fails: ctx 0x%X, crx 0x%X\n", ctx, crx);
      printf("\nERROR: %s(), line %d: Exit ...\n", __FUNCTION__, __LINE__);
getchar(); exit(1);
   }
}


void concatChar_C_late(unsigned char* buf, unsigned long long* C_late_p)
{
   *C_late_p = 0;

   for (int i=0, j=EHA_BYTE_SIZE-1; i<EHA_BYTE_SIZE; i++, j--)
   {
      (*C_late_p) |= ((unsigned long long)(buf[i])) << (j*8);
   }
}


void concatChar_C_trans(unsigned char* buf, unsigned long long* C_trans_p)
{
   *C_trans_p = 0;

   for (int i=0, j=TRANS_COUNTER_BYTE_SIZE-1; i<TRANS_COUNTER_BYTE_SIZE;
i++, j--)
   {
      (*C_trans_p) |= ((unsigned long long)(buf[i])) << (j*8);
   }
}


void ld_md(unsigned int dcm_id, unsigned int mval, unsigned int dval)
{
   unsigned char ctx, crx;
   unsigned int mval_m1 = mval - 1;
   unsigned int dval_m1 = dval - 1;
   bool cond1 = (mval >= DCM_M_LOW) && (mval <= DCM_M_HIGH);
   bool cond2 = (dval >= DCM_D_LOW) && (dval <= DCM_D_HIGH);

   if (!cond1 || !cond2)
   {
      printf("\nld_md() ERROR: mval/dval out of bound!\n");
      printf("\nERROR: %s(), line %d: Exit ...\n", __FUNCTION__, __LINE__);
getchar(); exit(1);
   }

   switch (dcm_id) {
      case DCM1:
         ctx = (unsigned char)((EXT_OP1 << OP_POS) |
(EXT_OP1_LD_M_D_DCM1));
         break;
```

```
      case DCM2:
         ctx = (unsigned char)((EXT_OP1 << OP_POS) | (EXT_OP1_LD_M_D));
         break;
      default:
         printf("\nERROR: %s(), line %d: Exit ...\n", __FUNCTION__,
__LINE__); getchar(); exit(1);
         break;
   }

   WriteComChar(ctx);
   ctx = (unsigned char)(mval_m1);
   WriteComChar(ctx);
   ctx = (unsigned char)(dval_m1);
   WriteComChar(ctx);

   switch (dcm_id) {
      case DCM1:
         ctx = (unsigned char)((EXT_OP1 << OP_POS) |
(EXT_OP1_LD_M_D_DCM1));
         break;
      case DCM2:
         ctx = (unsigned char)((EXT_OP1 << OP_POS) | (EXT_OP1_LD_M_D));
         break;
      default:
         printf("\nERROR: %s(), line %d: Exit ...\n", __FUNCTION__,
__LINE__); getchar(); exit(1);
         break;
   }

   crx = ReadComChar();
   if (crx == ctx)
   {
#ifdef DEBUG_PRINT_SUCCEED_CONFIRMATION
      printf("\nld_md() succeeds: ctx 0x%X, crx 0x%X\n", ctx, crx);
#endif
      return;
   }
   else
   {
      printf("\nld_md() fails: ctx 0x%X, crx 0x%X\n", ctx, crx);
      printf("\nERROR: %s(), line %d: Exit ...\n", __FUNCTION__, __LINE__);
getchar(); exit(1);
   }
}


unsigned int readFileMD(bool maxChoice, bool freqMode, unsigned int** M_p,
unsigned int** D_p, double** clk_freq_p) {
   char filename[1024];
   unsigned int MDarraySize;

   if (maxChoice == MAXSPEED) {
```

```
      if (freqMode == LOW_FREQ_MODE) {
         strcpy(filename,"ML506_MAXSPEED_CLKIN_33M_CLKFX_LOWFREQ.csv");
         MDarraySize = MAXSPEED_LOWFREQ_SIZE;
      }
      else {
         strcpy(filename,"ML506_MAXSPEED_CLKIN_33M_CLKFX_HIGHFREQ.csv");
         MDarraySize = MAXSPEED_HIGHFREQ_SIZE;
      }
   }
   else {
      if (freqMode == LOW_FREQ_MODE) {
         strcpy(filename,"ML506_MAXRANGE_CLKIN_33M_CLKFX_LOWFREQ.csv");
         MDarraySize = MAXRANGE_LOWFREQ_SIZE;
      }
      else {
         strcpy(filename,"ML506_MAXRANGE_CLKIN_33M_CLKFX_HIGHFREQ.csv");
         MDarraySize = MAXRANGE_HIGHFREQ_SIZE;
      }
   }

   FILE *file = fopen(filename,"r");

   if (file == NULL) {
      printf("\nreadFileMD() ERROR: No such file or directory\n\n");
      printf("\nERROR: %s(), line %d: Exit ...\n", __FUNCTION__, __LINE__);
getchar(); exit(1);
   }

   *M_p = (unsigned int*)malloc(sizeof(unsigned int)*MDarraySize);
   *D_p = (unsigned int*)malloc(sizeof(unsigned int)*MDarraySize);
   *clk_freq_p = (double*)malloc(sizeof(double)*MDarraySize);

   char line[1024];
   char* record;
   unsigned int i = 0;

   while ( fgets(line, sizeof line, file) != NULL ) {
     fputs(line,stdout);
     record = strtok(line,",");
     printf("record : %s\n",record);
     *(*M_p + i) = atoi(record);
     record = strtok(NULL,",");
     printf("record : %s\n",record);
     *(*D_p + i) = atoi(record);
     record = strtok(NULL,",");
     printf("record : %s\n",record);
     *(*clk_freq_p + i) = atof(record);
     ++i ;
   }

   printf("\n\n*******************************\n\n");
   for (i=0; i<MDarraySize; i++)
```

```
      printf("M=%d, D=%d, clk_freq=%.8f\n", *(*M_p + i), *(*D_p + i),
*(*clk_freq_p + i));

   fclose(file);
   return MDarraySize;
}


void free2DArray_double(double** a, int row_size) {
    for (int i = 0; i < row_size; ++i) {
        free(a[i]);
    }
    free(a);
}


unsigned int readFileMD_2DCMs(bool maxChoice, bool freqMode, unsigned int**
DCM1_M_p, unsigned int** DCM1_D_p, double** DCM1_clk_freq_p, unsigned int**
DCM2_M_p, unsigned int** DCM2_D_p, double** DCM2_clk_freq_p) {
   char filename[1024];
   unsigned int MDarraySize;

   if (maxChoice == MAXSPEED) {
      if (freqMode == LOW_FREQ_MODE) {

strcpy(filename,"CascadeDCM_ML506_MAXSPEED_CLKIN1_33M_LOWFREQ.csv");
         MDarraySize = CASCADE_2DCMs_MAXSPEED_LOWFREQ_SIZE;
      }
      else {
         printf("\nERROR: %s(), line %d: Exit ...\n", __FUNCTION__,
__LINE__); getchar(); exit(1);
      }
   }
   else {
      if (freqMode == LOW_FREQ_MODE) {
         printf("\nERROR: %s(), line %d: Exit ...\n", __FUNCTION__,
__LINE__); getchar(); exit(1);
      }
      else {
         printf("\nERROR: %s(), line %d: Exit ...\n", __FUNCTION__,
__LINE__); getchar(); exit(1);
      }
   }

   FILE *file = fopen(filename,"r");

   if (file == NULL) {
      printf("\nreadFileMD() ERROR: No such file or directory\n\n");
      printf("\nERROR: %s(), line %d: Exit ...\n", __FUNCTION__, __LINE__);
getchar(); exit(1);
   }
```

```
   *DCM1_M_p = (unsigned int*)malloc(sizeof(unsigned int)*MDarraySize);
   *DCM1_D_p = (unsigned int*)malloc(sizeof(unsigned int)*MDarraySize);
   *DCM1_clk_freq_p = (double*)malloc(sizeof(double)*MDarraySize);

   *DCM2_M_p = (unsigned int*)malloc(sizeof(unsigned int)*MDarraySize);
   *DCM2_D_p = (unsigned int*)malloc(sizeof(unsigned int)*MDarraySize);
   *DCM2_clk_freq_p = (double*)malloc(sizeof(double)*MDarraySize);


   char line[1024];
   char* record;
   unsigned int i = 0;

   while ( fgets(line, sizeof line, file) != NULL ) {

#ifdef FAST_SWEEP_START_INDEX
     if (i < FAST_SWEEP_START_INDEX) {
        ++i;
        continue;
     }
#endif

     fputs(line,stdout);

     record = strtok(line,",");
     printf("record : %s\n",record);
     *(*DCM1_M_p + i) = atoi(record);
     record = strtok(NULL,",");
     printf("record : %s\n",record);
     *(*DCM1_D_p + i) = atoi(record);
     record = strtok(NULL,",");
     printf("record : %s\n",record);
     *(*DCM1_clk_freq_p + i) = atof(record);

     record = strtok(NULL,",");
     printf("record : %s\n",record);
     *(*DCM2_M_p + i) = atoi(record);
     record = strtok(NULL,",");
     printf("record : %s\n",record);
     *(*DCM2_D_p + i) = atoi(record);
     record = strtok(NULL,",");
     printf("record : %s\n",record);
     *(*DCM2_clk_freq_p + i) = atof(record);

     ++i ;
   }

   printf("\n\n*******************************\n\n");

#ifdef FAST_SWEEP_START_INDEX
   for (i=FAST_SWEEP_START_INDEX; i<MDarraySize; i++)
#endif
```

```
#ifndef FAST_SWEEP_START_INDEX
   for (i=0; i<MDarraySize; i++)
#endif

      printf("DCM1: M=%d, D=%d, clk_freq=%.8f;   DCM2: M=%d, D=%d,
clk_freq=%.8f\n", *(*DCM1_M_p + i), *(*DCM1_D_p + i), *(*DCM1_clk_freq_p +
i), *(*DCM2_M_p + i), *(*DCM2_D_p + i), *(*DCM2_clk_freq_p + i));

   fclose(file);
   return MDarraySize;
}


void main() {

time_t global_start_t4, global_end_t4;  double global_total_t_second4;
global_start_t4 = time(NULL);

char varFilename[1024];

sprintf(varFilename, "result\\global_timing_%d_%d.txt", FILE_START_INDEX,
FILE_START_INDEX + NUM_OF_TRIALS -1);

#if 0
FILE *f_global_timing = fopen(varFilename, "w");
if (f_global_timing == NULL) {
    printf("\nERROR: %s(), line %d: Exit ...\n", __FUNCTION__, __LINE__);
getchar(); exit(1);
}
#endif



for (int file_i=FILE_START_INDEX; file_i<(FILE_START_INDEX +
NUM_OF_TRIALS); file_i++) {

   time_t global_start_t, global_end_t;
   double global_total_t_second;
   global_start_t = time(NULL);

   clock_t start_t1, end_t1, total_t_ms1, total_t_second1;
   clock_t start_t2, end_t2, total_t_ms2, total_t_second2;
   clock_t start_t3, end_t3, total_t_ms3, total_t_second3;
   clock_t start_t4, end_t4, total_t_ms4, total_t_second4;
   clock_t start_t5, end_t5, total_t_ms5, total_t_second5;
   clock_t start_t6, end_t6, total_t_ms6, total_t_second6;
   clock_t start_t7, end_t7, total_t_ms7, total_t_second7;
   clock_t start_t8, end_t8, total_t_ms8, total_t_second8;

   time_t global_start_t1, global_end_t1;  double global_total_t_second1;
   time_t global_start_t2, global_end_t2;  double global_total_t_second2;
```

```
   time_t global_start_t3, global_end_t3;  double global_total_t_second3;

   sprintf(varFilename, "result\\puf_result_%d.csv", file_i);
   FILE *f = fopen(varFilename, "w");
   if (f == NULL) {
       printf("\nERROR: %s(), line %d: Exit ...\n", __FUNCTION__,
__LINE__); getchar(); exit(1);
   }

   sprintf(varFilename, "result\\timing_%d.txt", file_i);

#if 0
   FILE *f_timing = fopen(varFilename, "w");
   if (f_timing == NULL) {
       printf("\nERROR: %s(), line %d: Exit ...\n", __FUNCTION__,
__LINE__); getchar(); exit(1);
   }
#endif

start_t1 = clock();

   unsigned int* DCM1_M_p;
   unsigned int* DCM1_D_p;
   double* DCM1_clk_freq_p;
   unsigned int* DCM2_M_p;
   unsigned int* DCM2_D_p;
   double* DCM2_clk_freq_p;
   unsigned int MDarraySize;



   MDarraySize = readFileMD_2DCMs(MAXSPEED, LOW_FREQ_MODE, &DCM1_M_p,
&DCM1_D_p, &DCM1_clk_freq_p, &DCM2_M_p, &DCM2_D_p, &DCM2_clk_freq_p);

#if 0
   printf("\n\n---------------------------------\n\n");
   for (unsigned int i=0; i<MDarraySize; i++) {
      printf("M1=%d, D1=%d;  M2=%d, D2=%d\n", DCM1_M_p[i], DCM1_D_p[i],
DCM2_M_p[i], DCM2_D_p[i]);
   }
   printf("\n");
#endif

end_t1 = clock();
total_t_ms1 = end_t1 - start_t1;
//fprintf(f_timing, "\n Read all M, D, & clk freq values from file = %d
ms\n", total_t_ms1);



start_t2 = clock();
```

```
   unsigned char
C_trans_buf[TRANS_COUNTER_BYTE_SIZE]={0xff,0x12,0x34,0x56,0x78};
   unsigned char C_late_buf[EHA_BYTE_SIZE]={0xff,0x12,0x34,0x56};
   unsigned long long C_late, C_trans;
   unsigned long long* C_late_p = &C_late;
   unsigned long long* C_trans_p = &C_trans;
   unsigned int mval, dval;
   double failure_rate;

   unsigned char C_late_buf_cont[EHA_BYTE_SIZE*FRD_TOT];
   unsigned char* C_late_buf_cont_p;


   int linear_i;
   double failure_rate_array_per_freq_i[NUM_OF_FRD_ROWS*NUM_OF_FRD_COLS];


   int dim1 = NUM_OF_FRD_ROWS*NUM_OF_FRD_COLS;
   int dim2 = MDarraySize;
   double ** failure_rate_array_per_step_i = (double
**)malloc(dim1*sizeof(double*));
   for (int i = 0; i<dim1; i++) {
      failure_rate_array_per_step_i[i] = (double *)
malloc(dim2*sizeof(double));
   }


   dim1 = NUM_OF_FRD_ROWS*NUM_OF_FRD_COLS;
   dim2 = MDarraySize;
   double ** abs_ratio_minus_0_25 = (double
**)malloc(dim1*sizeof(double*));
   for (int i = 0; i<dim1; i++) {
      abs_ratio_minus_0_25[i] = (double *) malloc(dim2*sizeof(double));
   }



   OpenCom();

end_t2 = clock();
total_t_ms2 = end_t2 - start_t2;
//fprintf(f_timing, "\n?????? (2D malloc, Opencom()) = %d ms\n",
total_t_ms2);




global_start_t1 = time(NULL);
   for (unsigned int step_i=0; step_i < (NUM_OF_STAGES_PER_RO + 1);
step_i++) {

start_t3 = clock();
```

```
global_start_t2 = time(NULL);



#ifdef FAST_SWEEP_START_INDEX
   for (unsigned int j=0; j<NUM_OF_FRD_ROWS*NUM_OF_FRD_COLS; j++) {
   for (unsigned int i=FAST_SWEEP_START_INDEX; i<MDarraySize; i++) {
     abs_ratio_minus_0_25[j][i] = 2.0;
   }
   }

   for (unsigned int j=0; j<NUM_OF_FRD_ROWS*NUM_OF_FRD_COLS; j++) {
   for (unsigned int i=FAST_SWEEP_START_INDEX; i<MDarraySize; i++) {
      failure_rate_array_per_step_i[j][i] = 2.0;
   }
   }

start_t4 = clock();
global_start_t3 = time(NULL);
   for (unsigned int freq_i=FAST_SWEEP_START_INDEX; freq_i<MDarraySize;
freq_i += FREQ_I_STEP_SIZE) {

      if (freq_i==FAST_SWEEP_START_INDEX) {
         //fprintf(f_timing, "\n step_i=%d   Starting freq_i=%d\n", step_i,
freq_i);
      }
#else
   for (unsigned int j=0; j<NUM_OF_FRD_ROWS*NUM_OF_FRD_COLS; j++) {
   for (unsigned int i=0; i<MDarraySize; i++) {
     abs_ratio_minus_0_25[i] = 2.0;
   }                        /* necessary init */
   }

   for (unsigned int j=0; j<NUM_OF_FRD_ROWS*NUM_OF_FRD_COLS; j++) {
   for (unsigned int i=0; i<MDarraySize; i++) {
      failure_rate_array_per_step_i[j][i] = 2.0;
   }                        /* necessary init */
   }

start_t4 = clock();
global_start_t3 = time(NULL);
   for (unsigned int freq_i=0; freq_i<MDarraySize; freq_i++) {

      if (freq_i==0) {
         //fprintf(f_timing, "\n step_i=%d   Starting freq_i=%d\n", step_i,
freq_i);
      }
#endif

start_t5 = clock();

      mval = DCM1_M_p[freq_i];
```

```
      dval = DCM1_D_p[freq_i];
      ld_md(DCM1,mval,dval);

      mval = DCM2_M_p[freq_i];
      dval = DCM2_D_p[freq_i];
      ld_md(DCM2,mval,dval);


      sccfg(CLR,31);

      if (step_i>0) {
         sccfg(SET, step_i-1);
      }

      clear_eha();

#if 0
      read_C_trans(C_trans_buf);
      read_C_late(C_late_buf);
      concatChar_C_trans(C_trans_buf, C_trans_p);
      concatChar_C_late(C_late_buf, C_late_p);
      printf("\nC_late = %llu;  C_trans = %llu;  C_late/C_trans = %f\n",
C_late, C_trans, (double)(C_late) / (double)(C_trans));
#endif

      sta_stofrd(START);
      Sleep(1);
      sta_stofrd(STOP);

end_t5 = clock();
total_t_ms5 = end_t5 - start_t5;
//fprintf(f_timing, "\n step_i=%d  freq_i=%d  -- Send M&D, Config,
clear_eha, START, STOP : %d ms\n", step_i, freq_i, total_t_ms5);




start_t6 = clock();

      read_C_trans(C_trans_buf);
      concatChar_C_trans(C_trans_buf, C_trans_p);
      read_C_late_continuously(C_late_buf_cont);

      linear_i = 0;
      for (unsigned int row=0; row<NUM_OF_FRD_ROWS; row++) {
      for (unsigned int col=0; col<NUM_OF_FRD_COLS; col++) {

         C_late_buf_cont_p = C_late_buf_cont + (linear_i*EHA_BYTE_SIZE);
         concatChar_C_late(C_late_buf_cont_p, C_late_p);

         failure_rate = (double)(C_late) / (double)(C_trans);
         failure_rate_array_per_freq_i[linear_i] = failure_rate;
```

```
         failure_rate_array_per_step_i[linear_i][freq_i] = failure_rate;

         printf("\nfile_i = %d\n",file_i);
         printf("step_i = %d;  freq_i = %d;  row = %d;  col = %d\n",
step_i, freq_i, row, col);
         printf("M1 = %d;  D1 = %d;  M2 = %d;  D2 = %d\n",
DCM1_M_p[freq_i], DCM1_D_p[freq_i], DCM2_M_p[freq_i], DCM2_D_p[freq_i]);
         printf("C_late = %llu\n", C_late);
         printf("C_trans = %llu\n", C_trans);
         printf("C_late/C_trans = %f\n", failure_rate);

         if (C_trans == 0) {
            printf("\nERROR: %s(), line %d: Exit ...\n", __FUNCTION__,
__LINE__); getchar(); exit(1);
         }
         else {
            abs_ratio_minus_0_25[linear_i][freq_i] = fabs(failure_rate -
0.25);
         }

         linear_i++;



      }
      }

end_t6 = clock();
total_t_ms6 = end_t6 - start_t6;
//fprintf(f_timing, "\n step_i=%d  freq_i=%d  -- Read back C_late &
C_trans, Calc, Entire FRD matrix : %d ms\n", step_i, freq_i, total_t_ms6);


start_t7 = clock();

      int num_of_frds_exceeds_threshold = 0;
      for (int i=0; i<NUM_OF_FRD_ROWS*NUM_OF_FRD_COLS; i++) {
         if (failure_rate_array_per_freq_i[i] >
FAILURE_RATE_STOP_THRESHOLD) {
            num_of_frds_exceeds_threshold++;
         }
      }

      if (num_of_frds_exceeds_threshold == NUM_OF_FRD_ROWS*NUM_OF_FRD_COLS)
{
         //fprintf(f_timing, "\n step_i=%d   Stopping freq_i=%d\n", step_i,
freq_i);
         break;
      }

end_t7 = clock();
total_t_ms7 = end_t7 - start_t7;
```

```
//fprintf(f_timing, "\n step_i=%d  freq_i=%d  -- Check if we need to stop
sweeping : %d ms\n", step_i, freq_i, total_t_ms7);

   }



end_t4 = clock();
total_t_ms4 = end_t4 - start_t4;
//fprintf(f_timing, "\n step_i=%d -- Entire for-loop of freq_i  : %d ms\n",
step_i, total_t_ms4);

global_end_t3 = time(NULL);
global_total_t_second3 = difftime(global_end_t3,global_start_t3);
//fprintf(f_timing, "\n step_i=%d -- Entire for-loop of freq_i   : %f s\n",
step_i, global_total_t_second3);


start_t8 = clock();

   linear_i = 0;
   for (unsigned int row=0; row<NUM_OF_FRD_ROWS; row++) {
   for (unsigned int col=0; col<NUM_OF_FRD_COLS; col++) {



      double min_abs_ratio_minus_0_25 = 2.0;
      int min_index = -1;


#ifdef FAST_SWEEP_START_INDEX
      for (unsigned int i=FAST_SWEEP_START_INDEX; i<MDarraySize; i++) {
#endif
#ifndef FAST_SWEEP_START_INDEX
      for (unsigned int i=0; i<MDarraySize; i++) {
#endif
         if (abs_ratio_minus_0_25[linear_i][i] < min_abs_ratio_minus_0_25)
{
            min_abs_ratio_minus_0_25 = abs_ratio_minus_0_25[linear_i][i];
            min_index = i;
         }
      }



      if ((min_abs_ratio_minus_0_25 > 1) || (min_index<0)) {
         printf("\nERROR: %s(), line %d: Exit ...\n", __FUNCTION__,
__LINE__); getchar(); exit(1);
      }


      fprintf(f, "%d,%d,%d,%d,%.8E,%.8f\n", row, col, step_i, min_index,
```

```
min_abs_ratio_minus_0_25, 1/(2*DCM2_clk_freq_p[min_index])*1000);
   #if 1
      printf("\n*************************\n");
      printf("file_i = %d\n",file_i);
      printf("Found:\n");
      printf("row = %d;  col = %d\n", row, col);
      printf("\ni = %d\n", min_index);
      printf("DCM1:  M = %d;  D = %d;  f_25%% = %.8f Mhz\n",
DCM1_M_p[min_index], DCM1_D_p[min_index], DCM1_clk_freq_p[min_index]);
      printf("DCM2:  M = %d;  D = %d;  f_25%% = %.8f Mhz;  delay = %.8f
ns\n", DCM2_M_p[min_index], DCM2_D_p[min_index],
DCM2_clk_freq_p[min_index], 1/(2*DCM2_clk_freq_p[min_index])*1000);
      printf("min_abs_ratio_minus_0_25 = %.8E\n",
min_abs_ratio_minus_0_25);
   #endif

   linear_i++;



   }
   }

end_t8 = clock();
total_t_ms8 = end_t8 - start_t8;
//fprintf(f_timing, "\n step_i=%d  -- Estimate delay : %d ms\n", step_i,
total_t_ms8);

end_t3 = clock();
total_t_ms3 = end_t3 - start_t3;
//fprintf(f_timing, "\n step_i=%d   -- Each iter of for-loop of step_i   :
%d ms\n",  step_i, total_t_ms3);

global_end_t2 = time(NULL);
global_total_t_second2 = difftime(global_end_t2,global_start_t2);
//fprintf(f_timing, "\n step_i=%d   -- Each iter of for-loop of step_i   :
%f s\n",  step_i, global_total_t_second2);

   }

global_end_t1 = time(NULL);
global_total_t_second1 = difftime(global_end_t1,global_start_t1);
//fprintf(f_timing, "\nEntire for-loop of step_i : %f s\n",
global_total_t_second1);

global_end_t = time(NULL);
global_total_t_second = difftime(global_end_t,global_start_t);
//fprintf(f_timing, "\nEntire program : %f s\n", global_total_t_second);
//fprintf(f_global_timing, "\n  Trial %d : %f s\n", file_i,
global_total_t_second);

   CloseCom();
```

```
   free(DCM1_M_p);
   free(DCM1_D_p);
   free(DCM1_clk_freq_p);
   free(DCM2_M_p);
   free(DCM2_D_p);
   free(DCM2_clk_freq_p);
   free2DArray_double(failure_rate_array_per_step_i,
NUM_OF_FRD_ROWS*NUM_OF_FRD_COLS);
   free2DArray_double(abs_ratio_minus_0_25,
NUM_OF_FRD_ROWS*NUM_OF_FRD_COLS);
   fclose(f);
   //fclose(f_timing);
   printf("\n\nfile_i = %d\n",file_i);
   printf("\n\nPASSED\n");
   printf("------------");

}

global_end_t4 = time(NULL);
global_total_t_second4 = difftime(global_end_t4,global_start_t4);
//fprintf(f_global_timing, "\nTotal Time : %f s\n",
global_total_t_second4);
//fclose(f_global_timing);
printf("\n\nALL TRIALS PASSED\n");
printf("----------------------------");


   return;

}
```

7.2.2 defs.h

```
//ML501: r6c20
//ML506: r6c26
//ML510: r11c31
#define NUM_OF_FRD_ROWS                          6 //11, 6
#define NUM_OF_FRD_COLS                          20 //31, 20, 26
#define HIGH_TEMPER                              0 /* 1=70C, 0=room
temper */
#define NUM_OF_TRIALS                            1

#define NOP                                      0x0
#define LD_COL_ADDR                              0x1
#define LD_ROW_ADDR                              0x2
#define SET_CFG                                  0x3
#define EXT_OP1                                  0x7
#define NODATA                                   0x0
#define EXT_OP1_CLEAR_CFG                        0x1
```

```
#define EXT_OP1_START_FRD                           0x2
#define EXT_OP1_STOP_FRD                            0x3
#define EXT_OP1_READ_EHA                            0x4
#define EXT_OP1_CLEAR_EHA                           0x5
#define EXT_OP1_LD_M_D                              0x6
#define EXT_OP1_DFS_FREQ_MODE_LOW                   0x7
#define EXT_OP1_DFS_FREQ_MODE_HIGH                  0x8
#define EXT_OP1_LD_M_D_DCM1                         0x9
#define EXT_OP1_READ_C_TRANS                        0xa
#define EXT_OP1_READ_EHA_CONTINUOUSLY               0xb

#define OP_MASK                                     0xE0
#define DATA_MASK                                   0x1F
#define OP_POS                                      5
#define CFG_MAX                                     31
#define RD_EHA_BUF_SIZE                             9
#define EHA_BYTE_SIZE                               4
#define TRANS_COUNTER_BYTE_SIZE                     5
#define DCM_M_LOW                                   2
#define DCM_M_HIGH                                  33
#define DCM_D_LOW                                   1
#define DCM_D_HIGH                                  32
#define NUM_OF_STAGES_PER_RO                        14
#define FRD_TOT                                     (NUM_OF_FRD_ROWS *
NUM_OF_FRD_COLS)

#define SET                                         true
#define CLR                                         false
#define START                                       true
#define STOP                                        false

#define MAXSPEED_LOWFREQ_SIZE                       263
#define MAXSPEED_HIGHFREQ_SIZE                      51
#define MAXRANGE_LOWFREQ_SIZE                       263
#define MAXRANGE_HIGHFREQ_SIZE                      51
#define LOW_FREQ_MODE                               true
#define HIGH_FREQ_MODE                              false
#define MAXSPEED                                    true
#define MAXRANGE                                    false

#define CASCADE_2DCMs_MAXSPEED_LOWFREQ_SIZE         23566

#define DCM1                                        1
#define DCM2                                        2


typedef enum {COL, ROW} loadcr_CHOICE_T;
#define COL_MAX                                     63
#define ROW_MAX                                     5

//debug option
//#define DEBUG_PRINT_SUCCEED_CONFIRMATION            1
```

```
//#define DEBUG_PRINT_SUCCEED_READCOMCHAR            1

//Test
#define FREQ_I_STEP_SIZE                          16
#define FILE_START_INDEX                          0
#define BAUD_RATE                                 57600

#if (HIGH_TEMPER == 1)
   #define FAILURE_RATE_STOP_THRESHOLD               0.25
   #if (NUM_OF_STAGES_PER_RO == 14)
      #if 1
         #define FAST_SWEEP_START_INDEX              13900
         #define INV_SEL                             0
         #define INV_SEL_EN                          1
      #elif 0
         #define FAST_SWEEP_START_INDEX              14000
         #define NONE_INV_SEL                        1
      #endif
   #endif

#else /* (HIGH_TEMPER == 0) -- room temper */
   #define FAILURE_RATE_STOP_THRESHOLD               0.25
   #if (NUM_OF_STAGES_PER_RO == 14)
      #if 1
         #define FAST_SWEEP_START_INDEX              14200
         #define INV_SEL                             0
         #define INV_SEL_EN                          1
      #elif 0
         #define FAST_SWEEP_START_INDEX              14000
         #define NONE_INV_SEL                        1
      #endif
   #elif (NUM_OF_STAGES_PER_RO == 7)
      #if 0
         #define FAST_SWEEP_START_INDEX              22000
         #define INV_SEL                             6
         #define INV_SEL_EN                          1
      #elif 0
         #define FAST_SWEEP_START_INDEX              22000
         #define NONE_INV_SEL                        1
      #else
         #define FAST_SWEEP_START_INDEX              22000
      #endif
   #endif
#endif
```

Bibliography

[1] D. Lim, "Extracting Secret Keys from Integrated Circuits," in *MS Thesis. Massachusetts Institute of Technology*, 2004.

[2] O. Kommerling and M. G. Kuhn, "Design principles for tamper-resistant smartcard processors," in *USENIX Workshop on Smartcard Technology*, 1999.

[3] B. Gassend, D. Clarke, M. v. Dijk and S. Devadas, "Controlled physical random functions," in *Proceedings of 18th Annual Computer Security Applications Conference*, 2002.

[4] B. Gassend, D. Clarke, M. v. Dijk and S. Devadas, "Silicon physical random functions," in *Proceedings of the Computer and Communication Security Conference*, 2002.

[5] 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 with identification and authentication applications," in *Proceedings of the IEEE VLSI Circuits Symposium*, 2004.

[6] I. Verbauwhede and R. Maes, "Physical(ly) Unclonable Functions: An introduction to Intrinsic PUFs," in *Great Lakes Symposium on VLSI (GLSVLSI)*, 2011.

[7] K. Lofstrom, W. R. Daasch and D. Taylor, "IC identification circuit using device mismatch," in *IEEE International Solid-State Circuits Conference*, 2000.

[8] P. Tuyls, G. Schrijen and B. Skoric, "Read-proof hardware from protective coatings," in *Cryptographic Hardware and Embedded Systems (CHES)*, 2006.

[9] D. Puntin, S. Stanzione and G. Iannaccone, "CMOS unclonable system for secure authentication based on device variability," in *European Solid-State Circuits Conference (ESSCIRC)*, 2008.

[10] D. D. E. Holcomb, W. P. Burleson and K. Fu, "Power-Up SRAM State as an Identifying Fingerprint and Source of True Random Numbers," in *IEEE Transactions on Computers*, 2009.

[11] J. Guajardo, S. S. Kumar, G. Schrijen and P. Tuyls, "FPGA Intrinsic PUFs and Their Use for IP Protection," in *Cryptographic Hardware and Embedded Systems (CHES)*, 2007.

[12] S. S. Kumar, J. Guajardo, R. Maes, G.-J. Schrijen and P. Tuyls, "Extended abstract: The butterfly PUF protecting IP on every FPGA," in *IEEE International Workshop on Hardware-Oriented Security and Trust (HOST)*, 2008.

[13] A. Krishna and S. Narasimhan, "MECCA: a robust low-overhead PUF using embedded memory array," in *Cryptographic Hardware and Embedded Systems (CHES)*, 2011.

[14] Y. Zheng, A. Krishna and S. Bhunia, "ScanPUF: Robust ultralow-overhead PUF using scan chain," in *Asia and South Pacific Design Automation Conference (ASP-DAC)*, 2013.

[15] Q. Chen, G. Csaba, P. Lugli, U. Schlichtmann and U. Ruhrmair, "The Bistable

Ring PUF: A new architecture for strong Physical Unclonable Functions," in *Hardware-Oriented Security and Trust (HOST)*, 2011.

[16] J. Lee, D. Lim, B. Gassend, G. E. Suh, M. v. Dijk and S. Devadas, "Extracting secret keys from integrated circuits," in *IEEE Transactions on Very Large Scale Integration Systems*, 2005.

[17] G. E. Suh and S. Devadas, "Physical Unclonable Functions for Device Authentication and Secret Key Generation," in *Design Automation Conference (DAC)*, 2007.

[18] S. Morozov, A. Maiti and P. Schaumont, "An Analysis of Delay Based PUF Implementations on FPGA," in *ARC*, 2011.

[19] U. R. Uhrmair, F. Sehnke, J. S. Olter, G. Dror, S. Devadas and J. Schmidhuber, "Modeling attacks on physical unclonable functions," in *ACM conference on Computer and Communications Security (CCS)*, 2010.

[20] C.-E. Yin and G. Qu, "Temperature-aware cooperative ring oscillator PUF," in *IEEE International Workshop on Hardware-Oriented Security and Trust (HOST)*, 2009.

[21] V. Vivekraja and L. Nazhandali, "Feedback Based Supply Voltage Control for Temperature Variation Tolerant PUFs," in *International Conference on VLSI Design*, 2011.

[22] S. S. Mansouri and E. Dubrova, "Ring oscillator physical unclonable function with multi level supply voltages," in *International Conference on Computer*

Design (ICCD), 2012.

[23] C.-E. Yin and G. Qu, "LISA: Maximizing RO PUF's secret extraction," in *IEEE International Symposium on Hardware-Oriented Security and Trust (HOST)*, 2010.

[24] M.-D. Yu and S. Devadas, "Secure and Robust Error Correction for Physical Unclonable Functions," in *IEEE Design & Test of Computers*, 2010.

[25] Y. Dodis, L. Reyzin and A. Smith, "Fuzzy extractors: how to generate strong keys from biometrics and other noisy data," in *Advances in Cryptology - Eurocrypt*, 2004.

[26] A. Maiti and P. Schaumont, "Improving the quality of a physical unclonable function using configurable ring oscillators," in *IEEE Field Programmable Logic and Applications (FPL)*, 2009.

[27] X. Xin, J.-P. Kaps and K. Gaj, "A Configurable Ring-Oscillator-Based PUF for Xilinx FPGAs," in *Euromicro Conference on Digital System Design*, 2011.

[28] S. Katzenbeisser, U. Kocabas, V. van der Leest, A. R. Sadeghi, G. J. Schrijen and C. Wachsmann, "Recyclable PUFs: logically reconfigurable PUFs," in *Journal of Cryptographic Engineering*, 2011.

[29] K. Kursawe, A. Sadeghi, D. Schellekens, B. Skoric and P. Tuyls, "Reconfigurable Physical Unclonable Functions - Enabling technology for tamper-resistant storage," in *IEEE International Workshop on Hardware-Oriented Security and Trust (HOST)*, 2009.

[30] D. Forte and A. Srivastava, "On improving the uniqueness of silicon-based physically unclonable functions via Optical Proximity Correction," in *Design Automation Conference (DAC)*, 2012.

[31] R. Kumar, H. Chandrikakutty and S. Kundu, "On improving reliability of delay based Physically Unclonable Functions under temperature variations," in *IEEE International Symposium on Hardware-Oriented Security and Trust (HOST)*, 2011.

[32] C. Helfmeier, C. Boit, D. Nedospasov and J.-P. Seifert, "Cloning Physically Unclonable Functions," in *IEEE International Symposium on Hardware-Oriented Security and Trust (HOST)*, 2013.

[33] M. Majzoobi, E. Dyer, A. Elnably and F. Koushanfar, "Rapid FPGA delay characterization using clock synthesis and sparse sampling," in *IEEE International Test Conference (ITC)*, 2010.

[34] J. Wong, P. Sedcole and P. Y. K. Cheung, "Self-characterization of Combinatorial Circuit Delays in FPGAs," in *International Conference on Field-Programmable Technology*, 2007.

[35] D. Merli, D. Schuster, F. Stumpf and G. Sigl, "Semi-invasive EM attack on FPGA RO PUFs and countermeasures," in *6th Workshop on Embedded Systems Security (WESS), ACM*, 2011.

[36] A. Maiti and P. Schaumont, "Research on Physical Unclonble Functions (PUFs) at SES Lab, VT," [Online]. Available: http://rijndael.ece.vt.edu/puf/main.html.

[37] A. Rukhin, J. Soto, J. Nechvatal, M. Smid, E. Barker, S. Leigh, M. Levenson, M. Vangel, D. Banks, A. Heckert, J. Dray, S. Vo and L. Bassham III, "A statistical test suite for random and pseudorandom number generators for cryptographic applications," NIST Special Publication 800-22 Revision 1a, 2010.

[38] C.-E. Yin and G. Qu, "Improving PUF security with regression-based distiller," in *Design Automation Conference (DAC)*, 2013.

[39] Xilinx, Inc., "Virtex-5 FPGA User Guide," 2012.

[40] Xilinx, Inc., "Constraints Guide," 2011.

[41] S. Bourdeauducq, "Simple RS232 UART," 2010. [Online]. Available: http://opencores.org/project,mmuart. [Accessed 2014].

[42] Microsoft Corporation, "Communications Resources," 2014. [Online]. Available: http://msdn.microsoft.com/en-us/library/windows/desktop/aa363196(v=vs.85).aspx. [Accessed 2014].

[43] KNJN LLC, "KNJN FX2 FPGA development boards," 2014. [Online]. Available: http://www.knjn.com/docs/KNJN%20FX2%20FPGA%20boards.pdf. [Accessed 2014].