Universidade Estadual de Campinas
Instituto de Computação

Jefferson Rodrigo Capovilla

Improving the Statistical Variability of Delay-based Physical Unclonable Functions

Otimização da Variabilidade Estatística em Circuitos Physical Unclonable Functions Baseados em Atraso

CAMPINAS
2016

Jefferson Rodrigo Capovilla

Improving the Statistical Variability of Delay-based Physical Unclonable Functions

Otimização da Variabilidade Estatística em Circuitos Physical Unclonable Functions Baseados em Atraso

Dissertação apresentada ao Instituto de Computação da Universidade Estadual de Campinas como parte dos requisitos para a obtenção do título de Mestre em Ciência da Computação.

Thesis presented to the Institute of Computing of the University of Campinas in partial fulfillment of the requirements for the degree of Master in Computer Science.

Supervisor/Orientador: Prof. Dr. Guido Costa Souza de Araújo
Co-supervisor/Coorientador: Prof. Dr. Mario Lúcio Côrtes

Este exemplar corresponde à versão final da Dissertação defendida por Jefferson Rodrigo Capovilla e orientada pelo Prof. Dr. Guido Costa Souza de Araújo.

CAMPINAS
2016

Agência(s) de fomento e nº(s) de processo(s): Não se aplica.

Ficha catalográfica
Universidade Estadual de Campinas
Biblioteca do Instituto de Matemática, Estatística e Computação Científica
Ana Regina Machado - CRB 8/5467

C173i

Capovilla, Jefferson Rodrigo, 1984-
Improving the statistical variability of delay-based Physical Unclonable Functions / Jefferson Rodrigo Capovilla. – Campinas, SP : [s.n.], 2016.

Orientador: Guido Costa Souza de Araújo.
Coorientador: Mario Lúcio Côrtes.
Dissertação (mestrado) – Universidade Estadual de Campinas, Instituto de Computação.

1. Funções físicas inclonáveis. 2. Variações do processo de fabricação. 3. Monte Carlo, Método de. 4. Circuitos integrados digitais. 5. Circuitos integrados - Integração em escala muito ampla. 6. Circuitos integrados - Integração em larga escala. 7. Criptografia de dados (Computação). I. Araújo, Guido Costa Souza de,1962-. II. Côrtes, Mario Lúcio,1950-. III. Universidade Estadual de Campinas. Instituto de Computação. IV. Título.

Informações para Biblioteca Digital

Título em outro idioma: Otimização da variabilidade estatística em circuitos Physical Unclonable Functions baseados em atraso
Palavras-chave em inglês:
Physical unclonable functions
Manufacturing process variations
Monte Carlo method
Digital integrated circuits
Integrated circuits - Very large scale integration
Integrated circuits - Large scale integration
Data encryption (Computer science)
Área de concentração: Ciência da Computação
Titulação: Mestre em Ciência da Computação
Banca examinadora:
Guido Costa Souza de Araújo [Orientador]
Carlos Alberto dos Reis Filho
Ricardo Pannain
Data de defesa: 18-03-2016
Programa de Pós-Graduação: Ciência da Computação

Universidade Estadual de Campinas
Instituto de Computação

INSTITUTO DE COMPUTAÇÃO

Jefferson Rodrigo Capovilla

Improving the Statistical Variability of Delay-based Physical Unclonable Functions

Otimização da Variabilidade Estatística em Circuitos Physical Unclonable Functions Baseados em Atraso

Campinas, 18 de março de 2016

Dedication

To my beloved family, Mario, Rose and Jessica. To my love, Daniela.

*“Not knowing it was impossible,
he did it.”*

(J. Rosseau)

Acknowledgements

I would like to first express my sincere gratitude to my research advisors, Prof. Guido Araújo and Prof. Mario Côrtes. This work would not be possible without their profound knowledge of VLSI design, sharp technical insights, and years after years of patient guidance. I also appreciate their effort in instructing me the techniques to write a scientific paper, and later in revising it. I can't thank Guido and Mario enough for always being supportive, making graduate school such a memorable adventure for me.

I'm also very grateful to Prof. Carlos Reis, Prof. Ricardo Pannain, Prof. Paulo Centoducatte and Prof. Frank Behrens for serving on my qualifying exam and dissertation committee. I am honoured to know that my work have been evaluated by reference professionals in the field of VLSI design, both in analog and digital perspective. Their sharp questions and critiques make me think deeper and in different perspectives about the challenges from the promising simulation result to real devices. In particular, Prof. Reis provided a deep analog analysis that should be considered when manufacturing the devices in silicon.

I would like also to thank the ones who helped me with the simulation infrastructure. Daniel Vidal, in setting up the simulation tools on external partition and performing some sanity tests; IT supporters of the Institute of Computing, who are always willing to help in my most peculiar difficulties. A special thank to LSC supporters, for working very hard in keeping all the necessary tools and remote access available.

The condition of doing the master's while working in a private company is a real challenge. So I would like to thank CPqD for encouraging me in keeping following the course, providing time for the weekly meetings, and being comprehensive on the moments I had to put extra effort on my research and during the dissertation.

Last, but not least, I would like to thank my loved ones, Mario, Rose, Jessica and Daniela, who have supported me throughout entire process, both by keeping me harmonious and helping me putting pieces together. I will be grateful forever for your love.

Resumo

Physical Unclonable Functions (PUFs) são circuitos que exploram da variabilidade estatística do processo de fabricação para criar uma identidade única para os dispositivos. PUFs são usados na construção de primitivas criptográficas para aplicações tais como autenticação, geração de chaves e proteção de propriedade intelectual. Devido ao seu baixo custo e simplicidade de construção, Arbiter PUFs (APUFs) baseados em atraso são considerados um mecanismo criptográfico em potencial para a integração em dispositivos IoT de baixo custo (e.g. tags RFID). Embora APUFs já foram alvo de diversas pesquisas, pouco avanço foi feito em se avaliar como melhorar a imprevisibilidade destes circuitos utilizando técnicas de desenvolvimento de circuitos VLSI. Esta dissertação utiliza biblioteca AMS 350nm e simulação Monte-Carlo em SPICE para analisar como a seleção apropriada do elemento árbitro e o redimensionamento de células podem afetar a variabilidade de atraso de APUFs. Os resultados experimentais mostram que a combinação do árbitro apropriado e redimensionamento das células podem melhorar consideravelmente a distribuição do Peso de Hamming nas respostas do APUF, assim resultando em um projeto mais confiável e menos tendencioso.

Abstract

Physical Unclonable Functions (PUFs) are circuits which exploit the statistical variability of the fabrication process to create a unique device identity. PUFs have been used in the design of cryptographic primitives for applications like device authentication, key generation and intellectual property protection. Due to its small cost and design simplicity, delay-based Arbiter PUFs (APUFs) have been considered a cryptographic engine candidate for the integration into low-cost IoT devices (e.g. RFID tags). Although APUFs have been extensively studied in the literature, not much work has been done in understanding how to improve its response variability using VLSI design techniques. This dissertation uses extensive AMS 350nm SPICE-based Monte-Carlo simulation to analyze how the proper selection of arbiter element and gate sizing can affect the delay variability of APUFs. Experimental results show that the combination of the appropriate arbiter and gate sizing configuration can considerably improve the Hamming Weight distribution of the APUF response, thus resulting in more reliable and less biased designs.

List of Figures

2.1 Generic delay PUF circuit 18
2.2 APUF mux circuit 18
2.3 Delay PUF circuit example (Chip 1) 19
2.4 Delay PUF circuit example (Chip 2) 20
2.5 PUF for authentication procedure 21
2.6 Transistor geometry [1] 22
2.7 Inter and Intra-die Variations [2] 23
2.8 Resistor geometry [1] 25
2.9 Transistor corner analysis [3] 26
2.10 Monte-Carlo Analysis [3] 27
2.11 Hamming Weight table 29

3.1 APUF tri-state buffer circuit 30
3.2 RC circuit 31
3.3 DFF internal architecture [1] 34
3.4 (a) Unbalanced SR-Latch;(b) Balanced SR-Latch 35
3.5 NAND CMOS 35

4.1 Cluster de simulação dos experimentos 37
4.2 Single cell characterization circuit 38
4.3 Transient result of the simulation 38
4.4 Standard deviation (σ_g) as gate strength increases. 39
4.5 Gate delay standard deviation σ_gs a function of K_W ($K_L = 1$) 40
4.6 Gate delay standard deviation σ_gs a function of K_L ($K_W = 1$) 40
4.7 Delay variability fast cell 40
4.8 Delay variability slow cell 41
4.9 Setup and Hold times specification for rising-edge-triggered flip-flop 42
4.10 DFF characterization procedure [2] 43
4.11 DFF arbiter characterization circuit 43
4.12 Setup and Hold times specification for SR-Latch 43
4.13 SR-Latch arbiter characterization circuit 44
4.14 SR-Latch arbiter characterization - setup time simulation 44
4.15 SR-Latch arbiter characterization -hold time simulation 45
4.16 Delay network circuit in ADE 47
4.17 Arbiter circuit in ADE 47
4.18 Simulation Framework 48
4.19 HWD comparison when using DFF and SR-Latch as arbiters 50
4.20 HWD comparison when using SR-Latch with different drive strength in delay network 50

A.1 Adding new library 60
A.2 Technology file for new library 60
A.3 Attach design library to technology file 60
A.4 Create new cellview 61
A.5 Adding a new instance of inverter gate 62
A.6 Adding a new instance of capacitor 62
A.7 Add pin 63
A.8 Inverter circuit ready 63
A.9 Create symbol from schematic 64
A.10 Add new instance of inverter circuit 65
A.11 Add vdd source 65
A.12 Add vpulse source 66
A.13 Add wire name 66
A.14 Testbench circuit ready 67
A.15 Setup model libraries 68
A.16 Set simulation variables 68
A.17 Configure plotted signals 68
A.18 Run Monte-Carlo Simulation 69
A.19 Monte-Carlo result 70
A.20 Copy inv cell 71
A.21 Copy inv cell window 71
A.22 Inv cell added to library 71
A.23 Original inv cell properties 72
A.24 Inv cell properties with K_L and K_W 72
A.25 Use the modified version of inv cell 73
A.26 Define K_L, K_W and run the simulation 73

List of Tables

2.1 CMOS process variation 25
2.2 Resistance according to process variation 26
2.3 Threshold tension of transistors 26

3.1 Truth tables 34

4.1 Average delay and standard deviation (σ_g) for combinational library gates 39
4.2 Mean and standard deviation for modified combinational gates 41
4.3 *Setup* or *Hold* times for the DFF and SR-Latch 45
4.4 Full circuit configurations 46
4.5 HW Mean and standard deviation for different configurations 51

Contents

1 Introduction 15

2 Basic concepts and related work 17

2.1 PUF definition . . . 17

2.1.1 Delay PUF . . . 18

2.2 Challenges and responses . . . 20

2.3 PUF properties . . . 21

2.4 Transistor Geometry and Process Variation . . . 22

2.5 Process Variability Simulation . . . 23

2.5.1 Corner Analysis . . . 24

2.5.2 Monte-Carlo Analysis . . . 26

2.6 PUF evaluation methodology . . . 28

3 Techniques to Improve Delay Variability 30

3.1 Delay network design techniques . . . 31

3.1.1 Simplified model of transistor delay . . . 31

3.1.2 Gate sizing . . . 32

3.1.3 Gate drive strength . . . 33

3.2 Arbiter design techniques . . . 33

4 Experimental Results 36

4.1 Simulation Infrastructure . . . 36

4.2 Delay Network Simulation . . . 37

4.2.1 Testbench for cell delay characterization . . . 37

4.2.2 Selecting the gate with best σ_g . . . 38

4.2.3 Variation of σ_p with gate sizing (ΔW and ΔL) . . . 38

4.3 Arbiter Simulation . . . 41

4.3.1 Selecting the arbiter with smallest forbidden window . . . 42

4.4 Full Circuit Simulation . . . 45

4.4.1 Testbench for Full Circuit Simulation . . . 46

4.4.2 Comparing HWD for different arbiters . . . 49

4.4.3 Improving HWD using gate sizing . . . 50

4.4.4 Overall result . . . 51

5 Conclusions and Future Work 52

Bibliography 54

A Tutorial: Monte-Carlo Simulation **57**

A.1 Environment Setup 57

A.2 Monte-Carlo Simulation of a 'Inverter' Gate 59

A.2.1 Creating new library and cellview 59

A.2.2 Circuit Design 61

A.2.3 Testbench Design 64

A.2.4 Running the simulation 64

A.3 Create and modify 'L' and 'W' of an inverter cell 70

A.4 Ocean script 73

B Tutorial: Monte-Carlo Simulation Framework **77**

Chapter 1

Introduction

Physical Unclonable Functions (PUFs) are circuits designed to exploit the statistical variations of its fabrication process to create a unique device identity. PUFs have been used in the design of cryptographic primitives for a variety of security applications like device authentication [4], key generation [5], remote enabling [6], among others.

Although they can be designed using physical parameters from different domains (optical, magnetic, acoustic, etc.) [7] silicon-based PUFs have become the technology of choice, given the maturity of its manufacturing process and the simplicity and low-cost of the resulting devices. Many silicon-based PUFs have been proposed, which exploit circuit parameter variations like: transistor threshold voltage (V_t) [8], voltage drop along power lines [9], capacitance layout variations [10], among others. Nevertheless, delay-based PUFs can still result in simple and small designs, the reason why they are the focus of this dissertation.

Delay-based PUFs were first introduced by Gassend et al. [6] and is based in a challenge and response behaviour. Challenge signals are applied to the input of the PUF, starting a race among different circuit paths in a *delay network*. As each path has a distinct delay, due to the statistical distribution of its resistances and capacitances, the response signals arrive at different moments at the end of the circuit paths, and are captured by an *arbiter*, which builds the PUF response output for the specific challenge. Circuit paths can be designed by using a combination of crossbars [6, 11] and tri-state buffers [12]. Because of the existence of an Arbiter, we will refer such PUFs as APUFs.

From a security perspective, PUFs have been under strong scrutiny as they can be target of a number of security attacks like: reverse engineering, delay parameter modelling, emulation and statistical modelling, side-channel attacks and machine learning [13–16]. Although there are still concerns about the overall PUF security for some applications domains, like RFID authentication, its simplicity and low-cost are very attractive design features [17]. PUF security rests on the difficulty to extract and model its behaviour as it depends on the variability of the intrinsic physical parameters of the fabrication process. For example, consider two identical PUF circuits. Although they have the exact same circuit layout they can produce different *output* when the same *input* vector is applied. This is due to the fact that PUFs are designed to amplify the small differences in its electrical parameters (resistance, capacitance etc) which result from the statistical variation of its physical parameters during the fabrication process (e.g. diffusion depth, dopant concentration, wire width, etc.) [18]. As a consequence, by only replicating its circuit layout, an attacker cannot clone a PUF. Moreover, notice also that the delay differences between two PUFs show up only when the circuit is stimulated. Hence any attack to a PUF aiming at capturing this difference will disturb its behavior and thus impact its response.

One of the most important features of a PUF circuit design is its ability to produce very distinct responses in different chips when the same challenge is applied to its input. The more distinct the responses are, the harder for an attacker to model its *challenge-response function* by monitoring its input/output. Hence, it is highly desirable to tailor the design of a PUF aiming at increasing its response variability. Nevertheless, although PUFs have been extensively studied in the literature, not much work has been done in understanding how gate sizing can be used to improve PUF responses. This dissertation uses extensive AMS 350 nm SPICE-based Monte-Carlo simulation to analyze the impact of gate sizing in the delay variability of Arbiter PUFs (APUF). It seeks to answer the following research questions: (1) Which design techniques can be used to increase the challenge-response variability of Arbiter PUFs? (2) And how these techniques can be used to produce well-balanced designs (in physical and statistical terms)?

To answer such questions, this dissertation gives the following contributions: (1) It proposes an approach which combines gate strength and channel length sizing to increase the delay variability of the APUF delay network; (2) It shows that the adoption of symmetric latch designs with reduced setup-time can improve the arbiter ability to detect small differences in the delay of the network paths.

The remainder of this dissertation is organized as follows. Section 2 discusses the APUF designs available in the literature and the workings of a typical APUF. Section 3 makes a thorough analysis on how the statistical delay distribution of APUFs varies as the size of its composing gates change. It also proposes an approach to use gate sizing to increase PUFs challenge-response variability. Section 4 discusses the experimental results that support the proposed approach. Section 5 concludes the work and describes future research.

Chapter 2

Basic concepts and related work

This section addresses the basic concepts required to fully understand the work described in this dissertation. It discusses PUF circuits (Section 2.1), challenge and response mechanisms (Section 2.2) and the required properties for a circuit to be considered a PUF (Section 2.3). It resumes the theory involving transistor design and lists the physical / electrical variations that occur during the manufacturing process (Section 2.4). Moreover it discusses the tools to simulate the designed circuits under the aforementioned variations (Section 2.5) and a method used to evaluate the results when comparing the proposed techniques(Section 2.6).

2.1 PUF definition

According to [7], *Physical Unclonable Functions* (PUFs) are circuits that map challenges to responses by exploiting intrinsic characteristics resulting from the fabrication process. They are designed to exploit such intrinsic characteristics, so that the same circuit produces different and unique responses when manufactured in different chips . Circuit responses can be considered an individual chip signature, which can be used for device identification (*device ID*).

There exist two main types of PUFs [7]: electronic and non-electronic. The difference consists in the nature of the components that contributes to the randomness which makes each PUF unique. Electronic components are typically used for the PUF part that performs measurement, processing and storage of the results.

The PUF used during this work belongs to the electronic category. The circuit randomness is based on the propagation time of two signals throw a set of configurable balanced paths, which are affected by variations during the manufacturing process. The path configurations are defined by the challenge input bits, and for each input, one response output bit is generated. Thus, a unique bit response can be generated by applying different challenges and grouping the response bits. The response bits that each chip produces can be used to uniquely identify them, similar as the fingerprint in human beings.

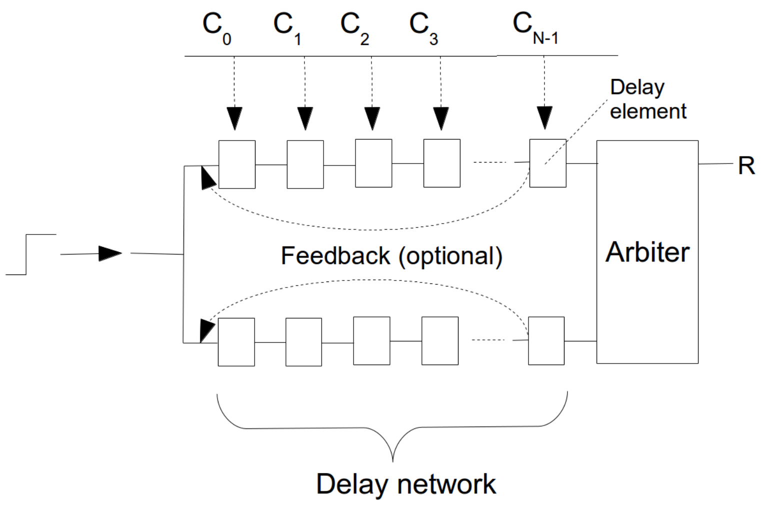


Figure 2.1: Generic delay PUF circuit

2.1.1 Delay PUF

Figure 2.1 shows a general model of delay PUF circuits. The variability of the circuit is attributed to the *delay network*, which exploits the variability of the different circuit paths (defined by the challenge C_i, $i = 0..n-1$) to create a signal race between the paths. In the case of ring oscillator PUFs [4], there exists also a presence of the *feedback* connection. The delay difference detection is performed by an *arbiter*, which uses storage elements (e.g. flip-flops, latches) to sample the output of the stage in order to produce the PUF response.

One of the fundamental requirements of an APUF design is to assure that all paths of the delay stage are exactly the same, i.e. every path from the stage input to its output is designed to have exactly the same nominal delay. Hence, the designer should work to guarantee that every path at each stage is a combination of gates and interconnects that produce the same nominal delay (in terms of static timing analysis). To achieve this goal the designer must have full control not only of the logic design process, but also of the placement and routing of the interconnects. This assures that any delay mismatch between two paths is created only by the variations in the manufacturing process and not by the design.

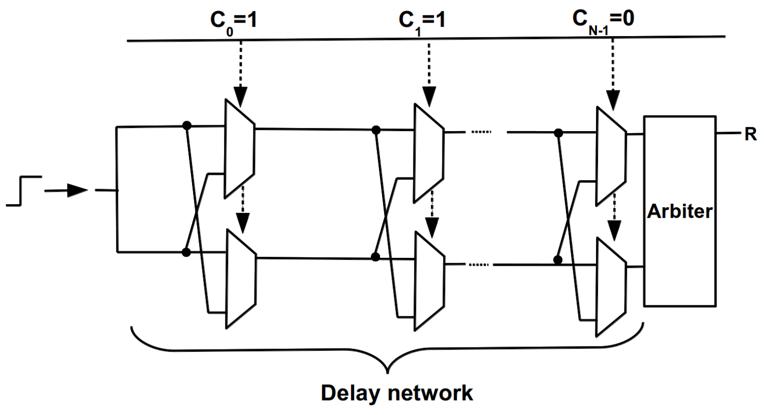


Figure 2.2: APUF mux circuit

The first APUF design was proposed by Lim et al. (Figure 2.2) [19]. Each stage of the delay network contains a crossbar switch, designed using two multiplexers. Notice that for this circuit, it is difficult to satisfy the aforementioned condition of having the exactly same nominal delay in the delay network, as the cross-connections are always longer than the straight ones. On the other hand, the architecture proposed by Ozturk e.t. all [12] works similarly as the former, but instead of crossing the paths, each signal has an exclusive path, and uses the variation on the delay gates as the source of randomness. It is totally balanced, both in terms of gates and interconnections, as shown in Figures 2.3 and 2.4. For this reason, the following work used Ozturk's architecture for basis.

Ozturk's circuit works as follows: first, one has to set up the paths in which the input signal will flow through the top and bottom paths. These paths are selected according to the applied challenge bits (Section 2.2) (C_0, C_1, C_{N-1}). Next, a step signal is applied at the input, which starts the race between the paths. Given that the circuit is well-balanced, the delay mismatch between the paths is exclusively due to the variations in the manufacturing process (Section 2.5). Figure 2.3 presents the behavior of a *chip 1* instance, in which the top path is faster than the bottom path. In this example, the top signal arrives in port 'Data' of the flip-flop after *10ns* while the bottom signal arrives in 'Clock' port after *12ns*, causing the registered value, named *response* (R), to be '1'. In opposite, chip 2 (Figure 2.4) presents a different behavior for the same input challenge but this time with the signal arriving earlier in 'clock' port, causing the response to be '0'. The APUF's arbiter, in Figure 2.2, is a single flip-flop which captures the race between the two racing signals at inputs D and Clk. The order in which the signals arrive at D and Clk will define if the response (R) to the challenge is "0" or "1".

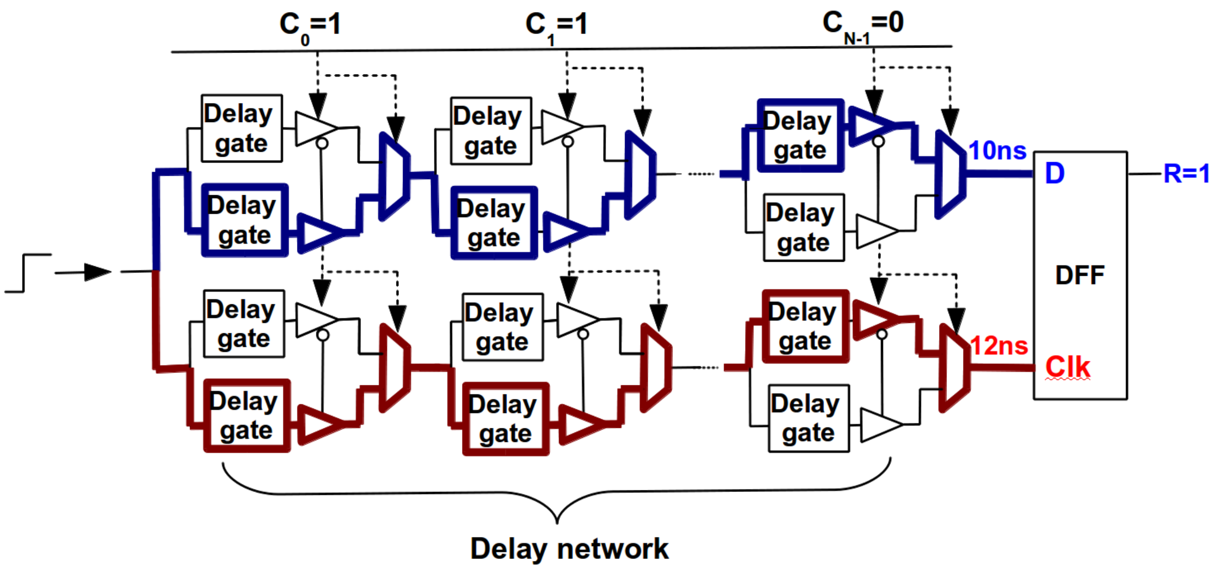


Figure 2.3: Delay PUF circuit example (Chip 1)

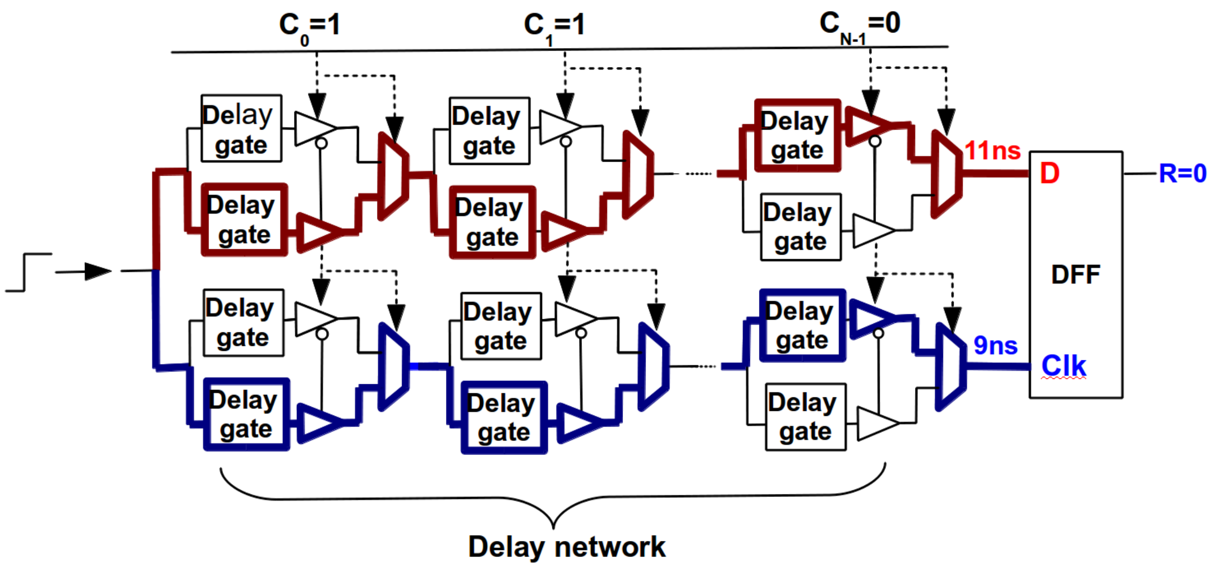


Figure 2.4: Delay PUF circuit example (Chip 2)

2.2 Challenges and responses

Consider again the APUF of Figure 2.2. It receives an input challenge vector $C_i, i = 0...N-1$ of N bits and generates a response bit R. Each stage of the APUF is a crossbar switch that selects which path the two racing signals will take at that stage. Crossbar switches can be designed using two multiplexers (MUXs) which take the challenge bit (C_i) as selectors. When $C_i = 0$ ($C_i = 1$) the crossbar operation is *pass through* (*crossed*). Remember that by construction the two crossbar paths should have the exact same static delay. The race starts at the first stage ($i = 0$) in which the two crossbar inputs are tied together and applied the same step signal. For example, in Figure 2.2 racing signals take the crossed path in the first and second stages ($C_{0,1} = 1$) and the pass through path in the last stage ($C_{N-1} = 0$).

The pair formed by an applied challenge and its respective response is named *challenge-response pair* (CRP). By applying multiples challenges and gathering the responses of the same chip instance, one builds the *CRP behaviour* of the instance. The CRP behaviour is the information that can be used to uniquely authenticate each device. The length of a CRP depends on the number of instances one intends to authenticate [19] and the resistance to brute force (trial and error) attacks [20]. Figure 2.5 demonstrates the usage of CRP as device authentication. The procedure, performed when the chips are already manufactured and encapsulated, is divided in two phases: chip characterization and chip authentication. The former is performed in a controlled ambient under safety requirements measures in place. A defined amount of chips are stimulated with the same collection of challenge bits, and their respective responses are stored in a common database, creating the CRP of each chip. For instance, the CRP of *chip 1* is called the *Database for chip 1*. The second phase occurs when the chip is assembled in an unsafe environment, being subject to various types of security attacks. The goal is to authenticate the device by comparing their current responses with the ones stored in its database, when the same challenges are applied. To avoid man-in-the-middle attacks [21], each CRP should be used only once, demanding for the database to have enough entries for the total number of authentication operations of each chip.

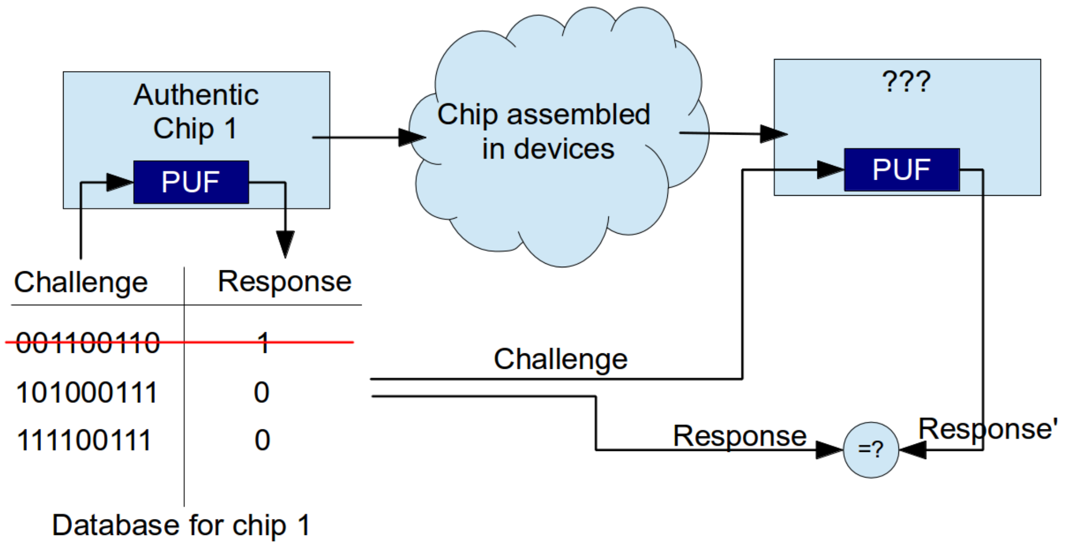


Figure 2.5: PUF for authentication procedure

2.3 PUF properties

This section describes the most important properties for a device to be considered a PUF based on the formal definition published by Maes et al in [7]. For the properties, consider the notation $\Pi : \chi \rightarrow \Upsilon : \Pi(x) = y$ as the representation of the challenge-response function of a determined PUF. Π represents the PUF function, χ the challenge set, Υ the response set, x a challenge instance from set χ and y a response instance from set Υ. The following PUF properties can then be defined based on this function. A PUF can be said to be:

1. "**Evaluatable:** given Π and x, it is **easy** to evaluate $y = \Pi(x)$". By *easy*, it means that the evaluation can be performed in polynomial time and effort, inducing little overhead on the overall system.

2. "**Unique:** $\Pi(x)$ contains some information about the identity of the physical entity embedding Π." By exploring this property using a well-defined set or population of PUF instantiations, at every challenge applied, the population is partitioned according to the response. Consecutive responses allow for smaller and smaller partitions until, optimally, a partition with a single PUF instantiation remains. In this case, the considered set of *challenge-response pairs* (CRPs) is sufficient to uniquely identify the PUF in the population. According to the population size, the aforementioned procedure might or might not be possible.

3. **Reproducible (or Robust):** For every value $y = \Pi(x)$, $y = y_0$ is the central value in a Gaussian probability distribution, and $y \neq y_0$ is a small variation around y_0. For this variation, error correction code algorithms (ECC) can be applied to recover

y_0. When this property is not present, it is possible that, for the same challenge, different results can be obtained. This property could be used to build true random number generators (TRNGs).

4. "**Unclonable:** given Π, it is hard to construct a procedure $\Gamma \neq \Pi$ such that $\forall x \in \chi : \Gamma(x) \approx \Pi(x)$ up to a small error." This is the core property of a PUF, which guarantees its usability as an identification device. For a PUF to be truly unclonable, it must be resistant to both the physical and mathematical approaches. *Physical resistance* is defined as the difficulty in creating a PUF $_\Gamma \neq \Pi$ (embedded in physically distinct entities) such that $\forall x :_\Gamma (x) \approx \Pi(x)$. Note that the hardness of producing a physical clone holds also for the manufacturer of the original PUF, reason why it is also classified as *manufacturer resistance*. The *Mathematical resistance* is applied if it is difficult to create an (abstract) mathematical procedure $f_\Gamma(x)$ such that $\forall x : f_\Gamma(x) \approx \Pi(x)$.

5. "**Unpredictable:** given only a set $Q = (xi, yi = \Pi(x_i)), i = 1...q$, it is hard to predict $y_c \approx \Pi(x_c)$ up to a small error, for x_c a random challenge, such that $(x_c, .) \notin Q$." If one can correctly predict the response of a given PUF for a random challenge by analyzing its previous CRPs, then it is possible to build a mathematical function with the same behaviour, thus breaking the previous property of unclonability.

6. "**One-way:** given only y and Π, it is hard to find $x \in \chi$ such that $\Pi(x) = y$." This is the same property commonly found in cryptographic *hash* functions. By knowing function Π (e.g. analyzing circuit layout) and the generated response y, it is infeasible to generate the input challenge x.

7. "**Tamper evident:** altering the physical entity embedding Π transforms $\Pi \rightarrow \Pi'$ such that with high probability $\exists x \in \chi : \Pi(x) \neq \Pi'(x)$ not even up to a small error." This property guarantees that if an attacker has physical access to the device, any attempt to monitor the internal paths will change the PUF properties, and thus, its behaviour.

2.4 Transistor Geometry and Process Variation

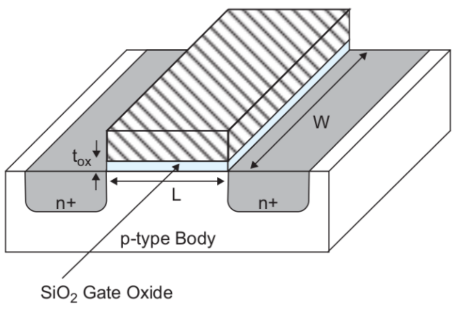


Figure 2.6: Transistor geometry [1]

Figure 2.6 shows a simplified model of a MOS (Metal-Oxide Semiconductor) geometry transistor [1] in which the source and drain are represented by the n-diffusion regions, and the gate is referenced by "SIO_2 Gate Oxide" arrow. The analog designer can have control in only two parameters related to the transistors geometry: channel length (L) and channel width (W). After a transistor is manufactured, its behaviour can vary at a function of different classes of process variations, as described in Boning, D.S and Nassif, S. [22]. From all the categories listed [22], the transistor is specially sensitive to *Device Geometry Variations*, which is divided in two sets:

Film thickness variations: Affects the gate oxide thickness. It is a critical variation but can be controlled within the same die. It has more impact between wafers, dies and lots.

Lateral dimension variations: Affects transistors lengths and widths. It is due to three main sources: photo-lithography proximity effect; mask, lens or photo system deviations; plasma etch dependencies.

2.5 Process Variability Simulation

With the sharp degree in feature length, process variability of integrated circuits has become the target of several research projects. As the process geometry is continuously shrinking, the ability to control all the parameters are becoming more and more difficult, and its impact in the circuit behaviour increasing proportionally.

Process variability can be classified in two groups: *inter-die* and *intra-die* variation. The *inter-die* variation occurs in different dies, such that for the same circuit, there are small differences between the dies of the same wafer, from different wafers and from different wafer lots. In the case of *intra-die*, the variations occur between different gates on the same die, and are also named *local variations* (Figure 2.7).

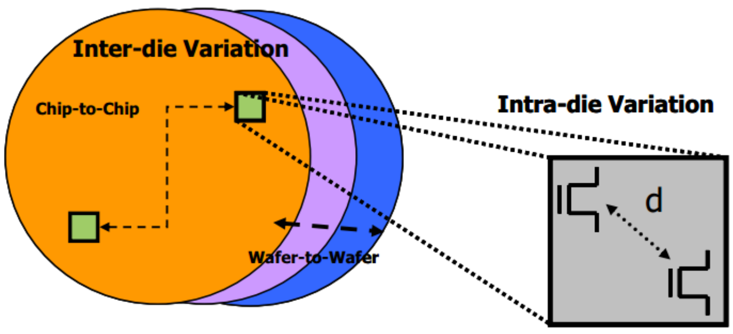


Figure 2.7: Inter and Intra-die Variations [2]

In digital design, process variation delay effects are modelled using *static timing analysis* (STA), based on three conditions: *worst case*, *typical case* and *best case*. The drawback of using this approach comes from the adoption of deterministic values, in which no statistical variation is present. Although this model provides good performance estimates for *inter-dies* variation, it cannot precisely model the statistical variations that occur for within the die.

For the design of delay based PUFs, it is mandatory that the used simulation tool can model as precise as possible the aforementioned variations. These variations need to be applied for each gate of the circuit layout, so that each of them presents a slightly different delay. The combination of the delay uncertainty at each gate is the source of entropy of the circuit. To simulate this scenario, two methods are commonly available: *Corner Analysis* and *Monte-Carlo Analysis*. The former, explained in Section 2.5.1, uses discrete conditions while the later, explained in Section 2.5.2, is based on statistical distributions.

Instead of performing the analysis using simulation tools, another option for this study would be the usage of FPGAs. We have discarded this possibility after our own analysis and previous works [23–25] enumerates several challenges to obtain a good performance under this environment: 1)regular cell placement to achieve symmetric PUF layout. 2)physical constraints on the FPGA fabric that forbids the designer to create complete symmetric routing paths. 3)limited number of gate types already manufactured on the FPGA, restricting the usage of geometry change technique in delay network(Section 3.1.2), and providing no guarantee if the arbiter circuit (SR-latch or DFF) is well-balanced(Section 3.2).

2.5.1 Corner Analysis

Corner analysis simulation is based on the fact that for each controlled parameter, the value of the electrical parameter in real silicon must lie between a minimum (MIN) and a maximum (MAX) quantity. This means that if the simulation process cover the MIN and MAX values of each parameter and the circuit still meets the required specification, then one can assume that the manufactured silicon will meet the specification too. The model files can be classified as follow:

- *Typical Mean (TM)*: Uses the typical value of each parameter.
- *Worst Speed (WS)*: Uses the worst value of each parameter, making the circuit slow.
- *Worst Power (WP)*: Uses the best value of each parameter, making the circuit fast, consequently power hungry.
- *Worst Zero (WZ)*: Consider differences between NMOS and PMOS regions, making the circuit slow for 'zero' transitions and fast for 'one' transitions.
- *Worst One (WZ)*: Consider differences between NMOS and PMOS regions, making the circuit slow for 'one' transitions and fast for 'zero' transitions.

For each process technology, the nominal value and variation range of each parameter is intensively monitored by the foundry, based on already manufactured samples. These data are provided to the designer via the so called technology library, in a format that can be loaded by the simulation tool.

As an example of corner analysis, consider the following example using a single resistor (Figure 2.8). The length L is the sum $L = L1 + L2 + L3$ and its nominal values described below:

- $W = 2\mu m$
- $L = 400\mu m$
- $R = 267k\Omega$

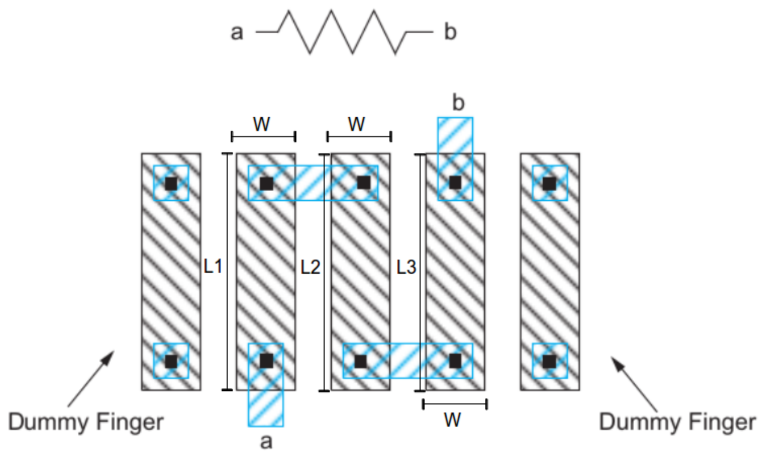


Figure 2.8: Resistor geometry [1]

For the process technology, consider the parameters *sheet resistance* R_{sh} and *effective width* when nominal W is set to 0.8 ($W_{eff|0.8}$). Range values for these parameters are shown in Table 2.1.

	Min	Typ	Max	
R_{sh}	1,0	1,2	1,4	$k\Omega/Sq.$
$W_{eff\|0.8}$	0.5	0.6	0.7	μm

Table 2.1: CMOS process variation

To calculate the impact of the process variation in the behaviour of each element, the technology library also contains equation models for the electrical characteristics of the components. To illustrate this procedure, consider the example of a resistor. Resistor's effective resistance (R_{eff}) is based on Eq. 2.1, being R_{sh} the sheet resistance, l and Δl respectively the length and length variation, and w and Δw respectively the width and width variation.

$$R_{eff} = R_{SH} * \frac{l - \Delta l}{w - \Delta w} \tag{2.1}$$

Table 2.2 shows the computation of the resistance according to the variation parameters. From this example, one can see that by using the corner analysis, the effective resistance can vary from $210k\Omega$ to $329k\Omega$.

The same procedure described above is also applicable to transistor models. For the transistor characterization shown in Table 2.3, the corner based simulation is performed according to Figure 2.9. This analysis is often too pessimistic, because it does not consider the parameter correlations. The ellipse region in Figure 2.9 delimits the effectively

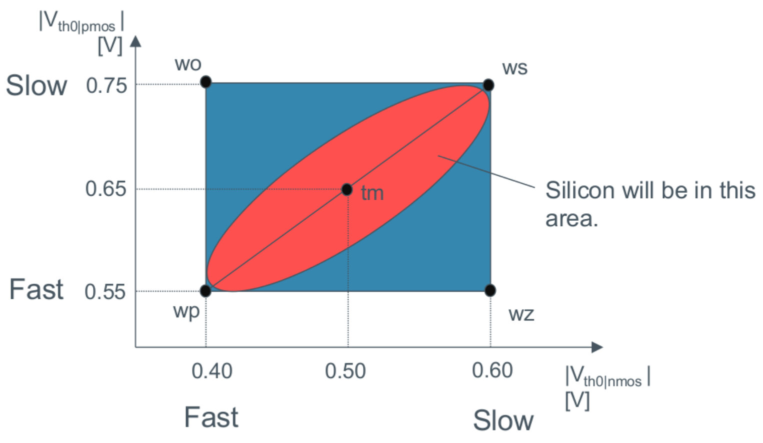
	R_{sh}	Δw	Δl	R_{eff}
WP	$1,0k\Omega/Sq.$	$0,1\mu m$	$0,0\mu m$	$210k\Omega$
TM	$1,2k\Omega/Sq.$	$0,2\mu m$	$0,0\mu m$	$267k\Omega$
WS	$1,4k\Omega/Sq.$	$0,3\mu m$	$0,0\mu m$	$329k\Omega$

Table 2.2: Resistance according to process variation

manufactured silicon characteristics. Another drawback of this type of simulation is that it does not consider the variations that occurs in a single chip between gates of the same type. For them, the same equation, conditions and variation range are applied.

	Min	Typ	Max	
$V_{th0\|nmos}$	0,4	0,5	0,6	V
$V_{th0\|pmos}$	-0,55	-0,65	-0,75	V

Table 2.3: Threshold tension of transistors



|V_{th0|pmos}| [V]
Slow
0.75
0.65
Fast
0.55
wo
ws
tm
wp
wz
Silicon will be in this area.
0.40
0.50
0.60
|V_{th0|nmos}| [V]
Fast
Slow

Figure 2.9: Transistor corner analysis [3]

2.5.2 Monte-Carlo Analysis

Researchers have shown that process variations cannot be avoided in silicon manufacturing process [26]. These variations are mostly related to statistical variations of the physical parameters of the materials (e.g. dopant density), equipment (e.g. lithography precision) and process (e.g. lateral diffusion depth) used in fabrication. As a result, the same transistor can show different performance due to variations in its final geometry (e.g. *Channel Length* (L)) and electrical parameters (e.g *RC delay*). In order to capture the impact of process variation in performance, statistical simulation models are applied, which can be used by commercial SPICE simulators (e.g. Cadence SPECTRE) to perform Monte-Carlo simulation of delay variability.

When using Monte-Carlo simulation, for each run every parameter is calculated randomly according to the distribution model provided by the foundry. Commonly, lot-to-lot, wafer-to-wafer and chip-to-chip variations are classified as "process variation", a single pseudo-random component from the perspective of individual chips, and modelled by an *uniform distribution*. Process variation is supposed to be similar for all transistors on the same chip. On the other hand, in-chip variations are decomposed into across-chip systematic and local random variations, which are modelled by independent *Gaussian distributions*.

As an example of Monte-Carlo analysis, consider a CMOS inverter circuit. For each transistor, the threshold voltage (V_{thOp}) is calculated based on Eq. 2.2, that is composed of three components: $V_{nominal}$ corresponds to the nominal threshold voltage of the transistor, *delvtop* represents the process variation parameters and *Dvthmat* corresponds to the *mismatch* variations. Figure 2.10 shows the region in which Monte-Carlo simulation estimates the parameters, and the density of probability of each region, known as *Standard Deviation* (or σ variation).

$$V_{thOp} = V_{nominal} + delvtop + dvthmat \tag{2.2}$$

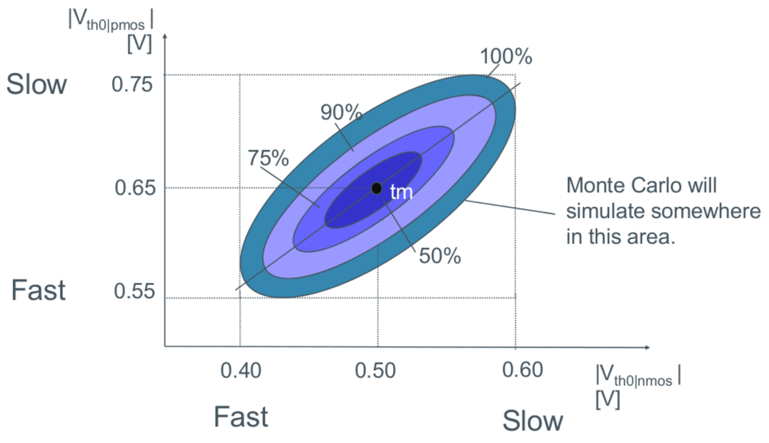


Figure 2.10: Monte-Carlo Analysis [3]

As a consequence of the variation in threshold and resistance, manufacturing processes affects the final delay of a designed cell. Ideally, the *delay network* circuit of Figure 2.1 should produce the same delay for any path created from the first to the last stage. As a consequence, the two racing paths should arrive at the same instant of time at ports D and Clk. However, due to the statistical variation of the fabrication process each of the gates that compose the paths have slightly different delays. The cumulative delay through all the stages will randomly change the arrival times of D and Clk, producing a desirable random response bit R. One possible drawback occurs when the arrival time of D and Clk do not respect the *setup/hold* time of the arbiter. In this case, *metastability* [1] can occur and the sample response R will be unpredictable. This is an undesirable feature, since the APUF should reproduce the same response when the same challenge in applied (*Reproducible* property - Section 2.3).

Given that the Monte-Carlo simulation offers a more realistic condition in terms of manufactured silicon chips, the rest of this work was performed using this kind of simulation.

2.6 PUF evaluation methodology

To apply PUF circuits in security solutions, its responses need to be reliably reproducible (robust) and also, unpredictable. A PUF is classified as robust when it generates similar responses for the same challenge applied several times, even under different operating conditions (temperature, supply voltage, noise level). The robustness of PUFs can be measured by mean of its bit error rate (Eq. 2.3), when compares the current response bitstring versus the bitstring obtained during the characterization process. The *hamming distance* (*HD*) between two bitstrings, is the number of bits that are different between each of them, and $|R_{ref}|$ is the length of the reference bitstring.

$$BER = \frac{HD(R_i, R_{ref})}{|R_{ref}|} \tag{2.3}$$

The unpredictability property is necessary to guarantee that an attacker cannot create a physical or mathematical module capable of correctly computing the response of an unknown challenge, from the knowledge of the previous responses of a given PUF. To evaluate if PUFs responses are biased, one would calculate the *Hamming Weight Distribution* (*HWD*) of its output and to verify the independence between PUFs instances, their *Hamming Distance* (*HD*) [16].

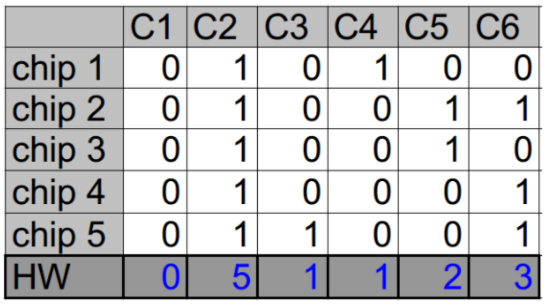
In this work, the experiments are performed using Spice-based simulations with deterministic models, which restrict the number of PUF samples and challenges due to the long simulation time and the deterministic results. As the robustness and independence evaluations (using *HD*) requires several samples and CRPs, these analysis are part of future work, when the PUF ASIC samples will be available. The focus of this work is to reduce the biasing in PUF responses, i.e, for the same challenge applied in different chips, there should be a balance between response bits '0'and '1', and thus it uses the *Hamming Weight* to evaluate the results.

Being n number of individuals and m the bitstring length, *Hamming Weight* (*HW*) [16] of a bitstring $X_i, i = 0...n-1$ is the number of non-zero bits of X_i, i.e. $HW(X_i) = \sum_{j=0}^{j=m} b_{ij}$, where b_{ij} is the value of bit j in the string X_i. The *Hamming Weight Distribution* (*HWD*) is an histogram which measures the frequency distribution of $HW(X_i)$ for all strings $X_i, i = 0..n-1$.

This measurement procedure is illustrated in the following example. In Figure 2.11(a), the table summarizes the responses obtained for each device (chip[1..5]) when different challenges (C[1..6]) are applied. The 'HW' row contains the sum of the responses obtained for each challenge, which is used to plot the *HWD*. Column 'C1' shows an example of biased '0' result, when every chip responded with '0'. Column 'C2' contains the behaviour when the result is biased towards '1'. Figure 2.11(b) shows the *HWD* of the particular example. Note that the graph contains a Gaussian distribution centered around '2'.

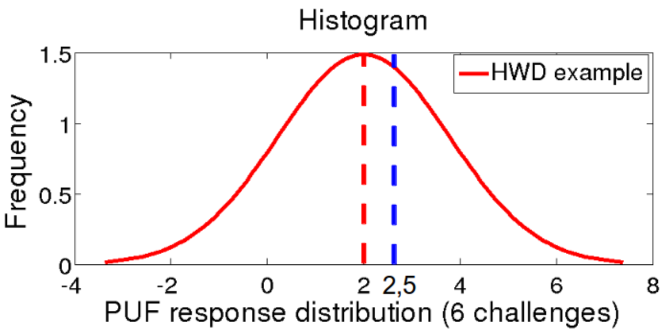
Ideally, the *HWD* of an APUF response should produce a Gaussian distribution [16] centered around half the number of chips driven by the same challenge ('2.5' in the above example). In such situation the APUF response is well balanced, i.e. the circuit efficiently exploits the process fabrication variability in such a way that the responses have no biasing towards '1's or '0's, which could eventually be exploited by an attacker. The more the

HWD of a PUF approaches this scenario the better the PUF is.



	C1	C2	C3	C4	C5	C6
chip 1	0	1	0	1	0	0
chip 2	0	1	0	0	1	1
chip 3	0	1	0	0	1	0
chip 4	0	1	0	0	0	1
chip 5	0	1	1	0	0	1
HW	0	5	1	1	2	3

(a)



(b)

Figure 2.11: Hamming Weight table

Although APUFs seem to represent a useful class of PUFs, it could become the target of reversing engineer modeling attacks [13]. Some techniques have been proposed to address this shortcoming, by obfuscating the PUF output, like arbiter designs based on an XOR array and feed forward arbiters [13]. Nevertheless, there are still questions concerning its resistance to such attacks [27]. The focus of this work is in using design techniques to improve the response variability of PUFs. Design techniques to improve the security of APUFs are not covered in this dissertation.

After describing the theory used in this work, the next chapters present a set of new design techniques for APUF designs and evaluate its performance focusing on answering the research questions presented in Section 1.

Chapter 3

Techniques to Improve Delay Variability

There are a number of ways to design an APUF delay stage. In [6], Gassend et al. uses a chain of cross-bar switches, built using muxes or tri-state gates, where the behaviour of each stage is defined by a bit of the challenge (Figure 2.2). Other types of delay-based PUFs are ring oscillators [4] and glitches [28]. Each such designs have their own advantages and drawbacks, but all of them could benefit from the gate sizing techniques proposed in this work. Hence, for the sake of simplicity, and without lacking generality, we have selected Ozturk's APUF [12] as the baseline design to discuss the techniques proposed herein.

In Ozturk's architecture, the delay paths at each stage (highlighted in Figure 3.1) are exclusive, i.e. each signal flows through its own path that contains tri-state buffers and delay gates. In order to avoid output spikes when both tri-state buffers are changing states (enabled to high-impedance), a MUX is also used at each path. The tri-state APUF is a good vehicle to analyze how gate sizing affects stage variability, given that it is a fully-balanced architecture (gates and net lengths) and enables the designer to easily change the delay gate type and configuration.

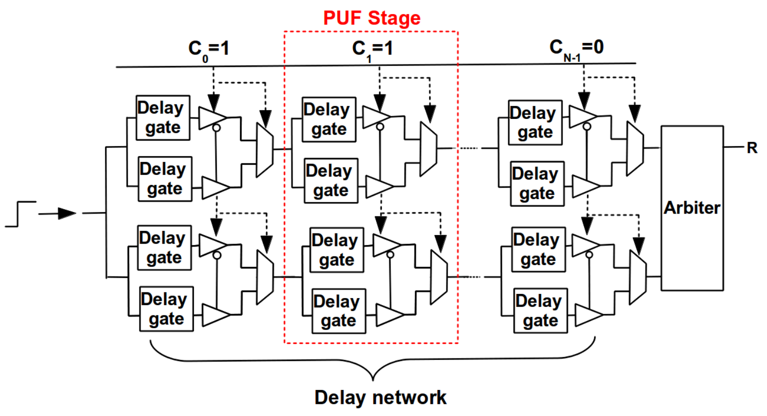


Figure 3.1: APUF tri-state buffer circuit

As described in the previous section, the APUF is a circuit which seeks to exploit, as much as possible, the variability of the fabrication process to create a physical unclonable function. The goal with this is that two identically designed APUFs could produce a different sequence of responses when the same sequence of challenges is applied.

Given that an APUF architecture is composed of two modules, a delay network and an arbiter, the designer should develop techniques which: (a) increase the delay variability of the delay network (Section 3.1); and (b) improve the ability of the arbiter to detect any small difference in the delay of the network paths which arrive at its inputs (Section 3.2). By doing so, the designer will contribute to increase the variability of the output response of the APUF. Notice that requirement (a) is an undesirable feature in a typical circuit design methodology. This work proposes a novel set of techniques to address these two goals. Overall, the experimental results using such techniques (Section 4) reveal that **slow delay networks and fast arbiters tend to produce better APUF designs**.

3.1 Delay network design techniques

Given that the path delay of an APUF network should expose the delay variations resulting from the fabrication process, the designer should focus in developing techniques that could expose, as much as possible, the intrinsic electrical variabilities. There are some potential ways in which a designer can achieve that, as described in Section 3.1.3 and Section 3.1.2. Section 3.1.1 contains a simplified model of transistor delay, used as a base to develop the design techniques. In this work the connection delay was not considered, focusing exclusively on the gates.

3.1.1 Simplified model of transistor delay

Transistor delay model

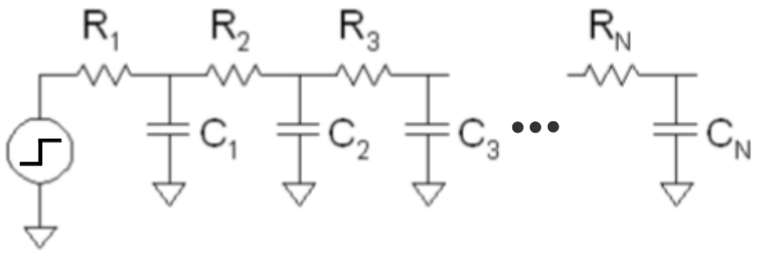


Figure 3.2: RC circuit

To model the impact of L, W and its variations in the transistor delay (τ) [1], consider that a circuit can be represented as an RC tree (circuit with no loops). The tree's root is the voltage source and the leaves are the capacitors at the end of the branches (Figure 3.2). The circuit delay can be estimated by *Elmore delay model* [1] as shown in Eq. 3.1. The resistance R is defined by Eq. 3.2 and the capacitance over the area (C_a) by Eq. 3.3.

$$\tau \propto [R * C] \tag{3.1}$$

$$R \propto [\frac{L}{W}] \tag{3.2}$$

$$C_a \propto [C_{ox} * L * W] \tag{3.3}$$

By replacing Eq. 3.2 and Eq. 3.3 into Eq. 3.1, the circuit delay is described as shown in Eq. 3.4. Notice there that the delay is directly affected by L, but not by W.

$$\tau \propto [R * C_a] = [R * \frac{L}{W}] * [C_{ox} * L * W] = R * L^2 * C_{ox} \tag{3.4}$$

The analysis below describes the impact of the process variation in each one of these delay parameters.

Transistor delay variation due to manufacturing process

To model a relative error, we consider a simple average error estimate, for example in equation $z = x * y$, the error is expressed as shown in Eq. 3.5.

$$\frac{\Delta z}{z} = \frac{\Delta x}{x} + \frac{\Delta y}{y} \tag{3.5}$$

Hence, from equation Eq. 3.4, the relative error is modelled by equation Eq. 3.6.

$$\begin{aligned}
\frac{\Delta\tau}{\tau} &= \frac{\Delta R}{R} + \frac{\Delta L}{L} + \frac{\Delta L}{L} + \frac{\Delta C_a}{C_a} \\
\Delta\tau &= (\frac{\Delta R}{R} + 2 * \frac{\Delta L}{L} + \frac{\Delta C_a}{C_a}) * (R * L^2 * C_a) \\
\Delta\tau &= \Delta R * L^2 * C + \Delta C_a * R * L^2 + 2\Delta L * R * L * C
\end{aligned} \tag{3.6}$$

Given that the variation of R and C_a are very small within the same die [29], the simplified delay equation can be modelled as in Eq. 3.7 below.

$$\Delta\tau = 2\Delta L * R * L * C \tag{3.7}$$

Therefore, in order to increase the delay variability of the transistor, the designer should increase the transistor's length given that the other parameters in Eq. 3.7 are not under his/her control.

3.1.2 Gate sizing

It is well-known that variations in the manufacturing process can impact the effective transistor geometries L and W. For example, mask variation and deposition fluctuations can create narrower or overstretched polysilicon. Moreover, sources and drains tend to diffuse laterally under the gate, producing a shorter effective channel length (L_{eff}). In [22], Boning et al. show that L_{eff} has the largest impact in the gate variability. Based on this and the delay model described in Section 3.1.1, we expect that any change in transistor's W barely affect the gate delay standard deviation. On the other hand, by increasing the size of L, the delay and the standard deviation of the gate could result in have a considerable increase.

It is important to define the difference between *variability* and *delay variability* when comparing the techniques on combinational gates. The variability is a function of the models provided by the foundry about the fabrication process for several parameter, which overall can be modelled in terms of *process variation* (variation between chips, wafers, lots), and *mismatch* (intra-chip variability). These variations are technology-dependent and are constant for every variation of L_{eff}. On the other hand, *delay variability* is a consequence of the manufacturing process, gate strength, and layout geometries. So, for the same process variation model, we expect the delay variability of the original geometry cell to be smaller when compared to the scaled in L versions.

The experiments related to gate sizing are described in Section 4.2.3.

3.1.3 Gate drive strength

Focusing on studying if the delay variability of the delay network varies accord to *gate drive strength*, we studied the behaviour of the combinational library cells. From [30], one can see that cells with high drive strength tend to produce fast rising and falling output slope. Assuming that slow transitions of output slopes produce higher output delay mean, we expected to see that the variation over the delay mean also increases. Thus, in theory the best candidates to increase delay variability are the combinational gates with weak drive strength. From them, it is also necessary to analyze which gate type/drive strength maximizes the *gate delay standard deviation* (σ_g).

Experiments to evaluate that are described in Section 4.2.2.

3.2 Arbiter design techniques

As described in Section 1, APUF arbiters are designed using storage elements, to capture the path delay differences resulting from the fabrication process variability. The racing condition on the two signals of the delay network, shown in Figure 3.1, starts with logic value '0' and ends with logic value '1' after applying a step signal. Therefore, an arbiter candidate needs to be either rising-edge triggered, or level-triggered with the extra requirement of keeping the stored value when both signals are '1'. From the aforementioned conditions, two possible arbiter candidates are the *D Flip-flop* (*DFF*) as proposed by Gassend et al. [6] or the NAND SR-Latch, suggested by Lang et al [31]. Their respective truth tables are shown in Table 3.1.

The arbiter element is used to record which of the racing edges arrived first, and quantize it in a digital value. Based on Figure 2.2, the racing signals are respectively connected to ports `CLK` and `D` of a (*DFF*). The value to be stored in the arbiter depends on the relative delay of the signals, being '1' when it arrives first in D, and '0' otherwise. In order to guarantee that the delay difference on the two paths at the inputs of an

Table 3.1: Truth tables

(a) SR NAND Latch

$\overline{S}$	$\overline{R}$	Action
0	0	not allowed
0	1	Q = 1
1	0	Q = 0
1	1	No Change

(b) D flip-flop

Clock	D	Q_{next}
Rising Edge	0	0
Rising Edge	1	1
Non-Rising	X	Q

arbiter is due only to the fabrication process, the designer must make sure that the two signals being compared go through similar (topologically and electrically) paths within the arbiter.

Arbiter logic paths

All internal logic paths of an arbiter, starting at its inputs to its output, should have the exact same delay. If there exists a tendency (bias) towards a certain value in the arbiter across many PUF instances, uniqueness will be reduced [4]. A careful analysis in the logic design of a typical CMOS DFF [1], shown in Figure 3.3, reveal that its internal paths are logically unbalanced and thus the path delays D-to-Q and CLK-to-Q are different. As the DFF inputs *D* and *Clk* have different functions, it is expected that they will be unbalanced. Therefore, by using a DFF as an APUF arbiter one cannot assure an unbiased evaluation of the delay differences of the paths arriving at its inputs, as required. Contrary to the DFF, the SR-Latch is a symmetric device (Figure 3.4b), and all its paths from input to output have the exact same delay [1], a required feature for correct APUF arbiter.

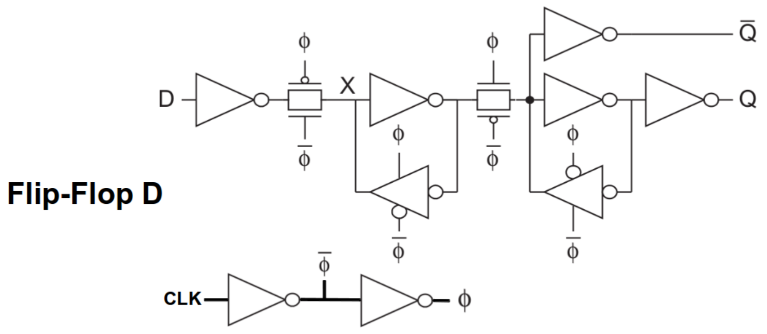


Figure 3.3: DFF internal architecture [1]

The architecture of a typical SR-LATCH, shown in Figure 3.4a, at first glance seems to be balanced, but a deeper analysis reveals that the feedback connections are not connected to the exact same NAND input ports. For this reason we propose a slightly different design in the SR-LATCH architecture, that changes the feedback connections to the same input ports (Figure 3.4b).

This scenario is better understood from the CMOS perspective. Considering the pull down NMOS transistors of the NAND gate (Figure 3.5), there are two parasitic capacitances connected in series, named q_1 and q_2. Depending on the order in which the input

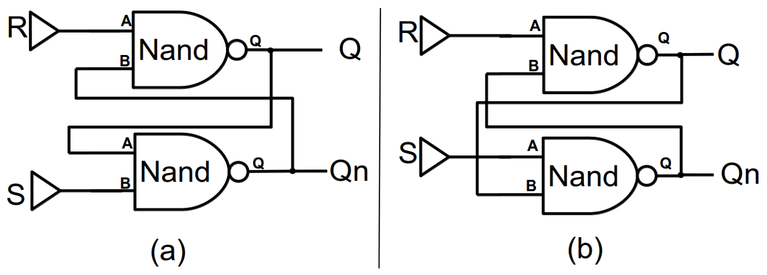


Figure 3.4: (a) Unbalanced SR-Latch;(b) Balanced SR-Latch

signals arrive, the capacitance discharge sequence is different. When input B changes to '1' before A, capacitance q_2 discharges while q_1 is still charging. Later when A arrives and changes to '1' it is only necessary to discharge q_1. The second scenario occurs when input B changes to '1' after A changes. In this case, only when $B = 1$ arrives capacitances are discharged through transistor B which drains a charge of $q_1 + q_2$, thus making the output signal slower .

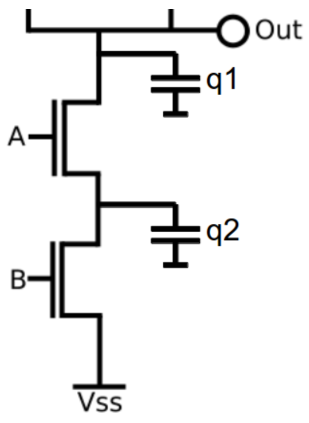


Figure 3.5: NAND CMOS

The impact of this design decision on HWD is confirmed by the experiments in Section 4.3.

Arbiter setup and hold times

Since the arrival times of the two signals at the arbiter inputs can be very close, metastability [1] can result at the arbiter output. In other words, given the small values for the path delay standard deviations (σ_p) at the arriving paths, it is possible that the setup and hold window restriction are not satisfied, thus resulting in an undefined value for Q. Moreover, given an arbiter with a large setup time, the circuit could also miss the delay difference of the two paths, producing a similar response as if they were the same, an undesirable situation for an arbiter. Hence, decreasing the t_{su} of the arbiter as much as possible is a way to assure that even small differences between the two path delays at its inputs will be detected. Moreover, if the delay difference between two paths is small enough and cannot satisfy the storage element *hold time* (t_h) the output transition is not guaranteed, another situation which should be avoided by the arbiter.

Section 4.3 evaluates the impact of using DFF and SR-Latch as APUF arbiters.

Chapter 4

Experimental Results

This section describes a number of experiments aiming at evaluating the impact of design techniques in APUF's response variability. The tri-state arbiter PUF, described in Section 1 is used as the baseline. Section 4.1 describes the simulation infrastructure used during the experiments. Section 4.2 describes the performed experiments focusing on finding the best logic design for the APUF delay network. In this section we evaluate the available cells in the standard library searching for the one that produce the largest delay variation. Also, we experimentally evaluate the impact of gate sizing (changes in transistor geometry - L and W) with respect to the gate delay variation. On Section 4.3 we experimentally define the forbidden window (setup + hold time) of each arbiter candidate(DFF, balanced and unbalanced SR-latch), searching for the architecture that exhibits the narrowest window, so as to minimize the chance of metastability. Thus we expect that even for small delay differences in the racing paths, the metastability problem will be reduced.

4.1 Simulation Infrastructure

Experiments used *Cadence Virtuoso Analog Design Environment (ADE) with Spice-based Monte-Carlo support*, a professional tool for analog circuit design and simulation. ADE supports a proprietary script language, named *OCEAN*, which gives the designer full control over the simulation process. Standard cells and process-related information were provided by the *AMS 350nm v3.80* hitkit.

The simulations performed in this work used a cluster (Figure 4.1) composed of seven nodes, each having a 32-cores CPU, 16GB RAM memory and individual hard-disks to store the results and to run the simulation locally. Nodes setup was done by installing the Cadence simulation tools natively, using a compatible operational system; no virtual machine was required. They are connected through a high-speed local LAN for fast result sharing. This robust infrastructure was required due to the high amount of SPICE analog simulations necessary to have a high enough sample space to perform an adequate statistical analysis.

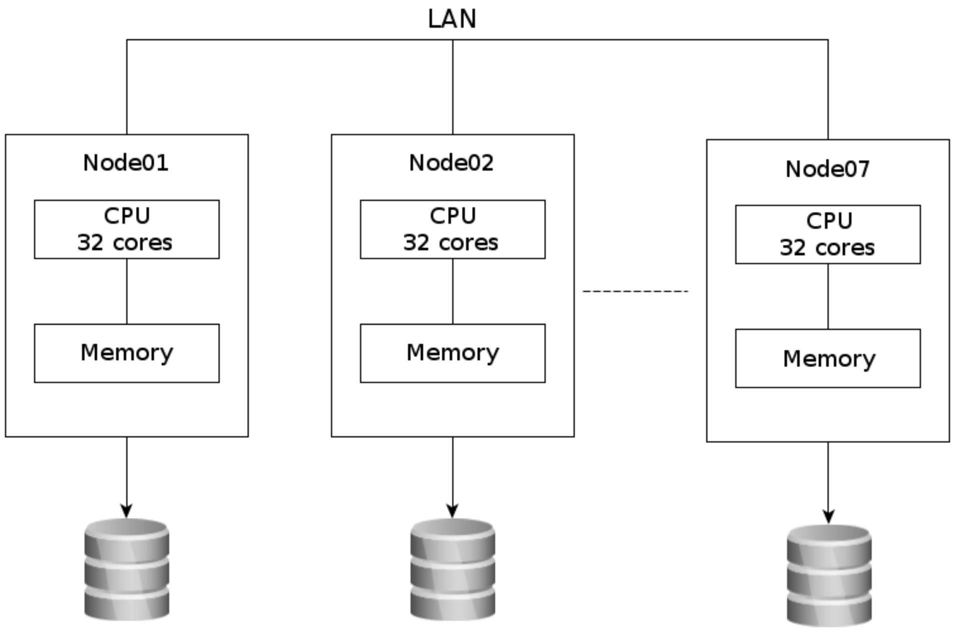


Figure 4.1: Cluster de simulação dos experimentos

4.2 Delay Network Simulation

A set of Monte-Carlo simulation experiments was performed in order to determine the best logic design for the APUF delay network as discussed in Section 3.1. This was achieved through a set of two experimental groups. In the first group (Section 4.2.2) the goal is to determine, from all gates in the standard-cell library, those which have the largest gate delay standard deviation (σ_g). The second group (Section 4.2.3) studies the dependence of σ_p with gate sizing, i.e. when the channel Length (L) and Width (W) of the gate transistors' change.

4.2.1 Testbench for cell delay characterization

Figure 4.2 shows an example of a circuit using $buf2$ cell, used to characterize every cell candidate for the delay network, whose results are listed in Section 4.2.2 and Section 4.2.3. The circuit is composed by three cells of the same type and drive strengths connected in series. The cell under characterization is located at the centre, so that the input signal slope and output load offer real conditions.

The characterization procedure starts by applying a step signal in the input port and measuring the delay difference between signals $out1$ and $out2$. As it is performed using analog simulation, the output signals are rising curves. The time measurement points used are located in 50% of VDD, in this case at $1,65V$. The cell delay mean and standard deviation uses Monte-Carlo simulation with 400 iterations, providing 400 pairs of ($out1$,$out2$) timestamps values, varied according to fabrication process distribution model (Figure 4.3). The difference of each pair of values provide the cell delay for the given Monte-Carlo iteration. These data are used to calculate cell delay mean and standard deviation. Table 4.1 shows the obtained results for each cell type.

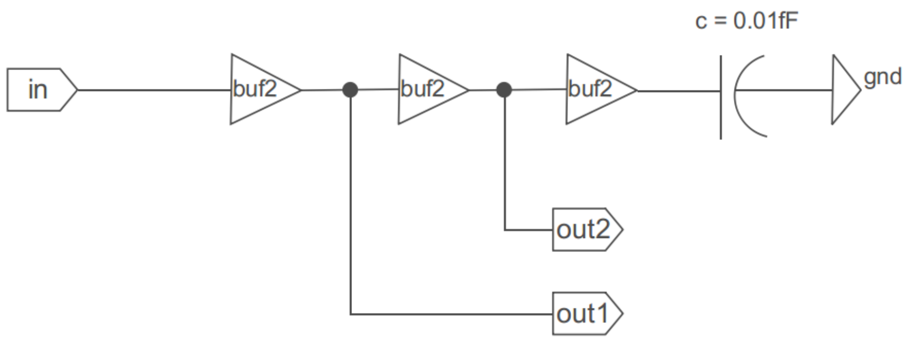


Figure 4.2: Single cell characterization circuit

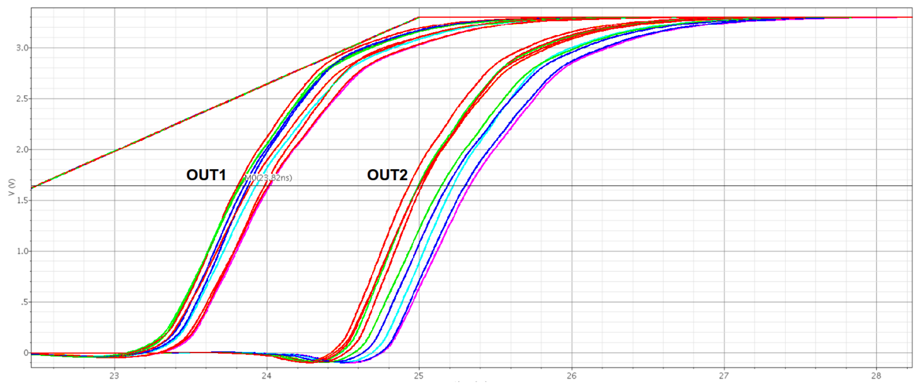


Figure 4.3: Transient result of the simulation

4.2.2 Selecting the gate with best σ_g

The AMS 350nm library contains a few drive strength options for each combinational gate. The objective of this study is to validate our hypothesis, described in Section 3.1.3, that the best candidate to increase delay variability is the one with the weakest drive strength. For this analysis, each gate type and strength went through Monte-Carlo simulation using 400 instances for each one. The results, shown in Figure 4.4, reveal that the σ_g of the library gates increase as the gate strength decreases. Hence, the weakest gates are those which produce the largest standard deviation over the delay mean. As shown in Table 4.1, for all combinational gate types and strengths of Figure 4.4, the `xor20` is the best candidate for the delay network, presenting the largest σ_g.

4.2.3 Variation of σ_p with gate sizing (ΔW and ΔL)

The experiment described in this section aims at analyzing the impact of gate sizing (changes in transistor geometry parameters – L and W) on σ_g and evaluate its consequence to the path delay deviation σ_p. The analytical model, described in Section 3.1.1, shows that the increase of σ_g is directly proportional to transistor L size, and that W size has a little impact over it.

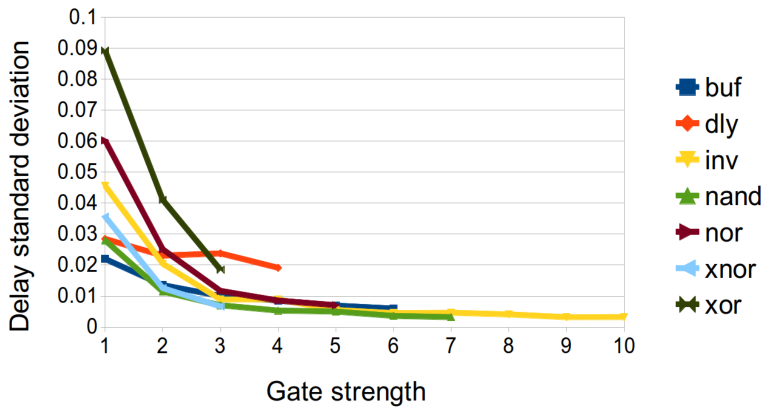


Figure 4.4: Standard deviation (σ_g) as gate strength increases.

Gate Type	Avg. Delay	Std. Dev. (σ_g)
buf2	2,718	0,022
dly12	3,826	0,028
inv0	6,966	0,046
nand20	5,762	0,028
nor20	7,005	0,060
xnor20	5,857	0,036
xor20	**10,295**	**0,089**

Table 4.1: Average delay and standard deviation (σ_g) for combinational library gates

To confirm this hypothesis, we respectively modify the original gate transistors size (L and W) by multiplying their baseline values L_0 and W_0 from the smallest gate by factors $K_L = L/L_0$ and $K_W = W/W_0$. Each version of the modified gate went through Monte-Carlo simulation using 400 instances, and the respective σ_g was determined.

Figure 4.5 shows the value of σ_g for various strengths of the combinational gates when K_W scales the size of W from 1 to 32 times. The results show that although σ_g increases only slightly for fairly small values of W, the variation is irrelevant. Thus, according to the analytical model in Section 3.1.1, increasing W will not affect much the standard deviation of the gate delay (σ_g) and therefore gates used in APUF network paths should have the smallest possible value of W.

A similar experiment was also performed for L. As shown in Figure 4.6, σ_g increases considerably with K_L. Therefore, contrary to W and according to the analytical model of Section 3.1.1, increasing the gate's transistors channel length has a significant impact on the gate delay standard deviation over the delay mean. Figure 4.7 shows the experimental result of the delay variation of an inverter gate when having the original L size, and Figure 4.8 when having it scale 32 times bigger.

The same behaviour was present in all other combinational gates of the library. Therefore, given the considerable impact of K_L in σ_g, *channel length sizing* will be used, from the remaining of this work, as the design technique of choice to control the σ_p of APUF delay network paths. Notice the reader that, contrary to a typical logic design, designing APUFs requires an increase in L so as to improve delay variability.

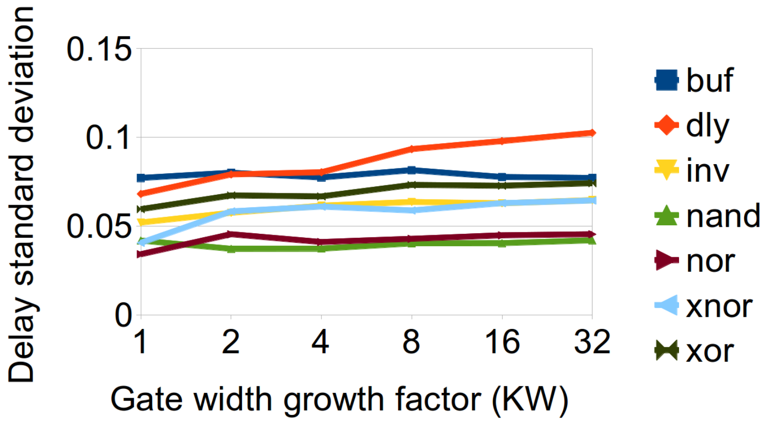


Figure 4.5: Gate delay standard deviation σ_gs a function of K_W ($K_L = 1$)

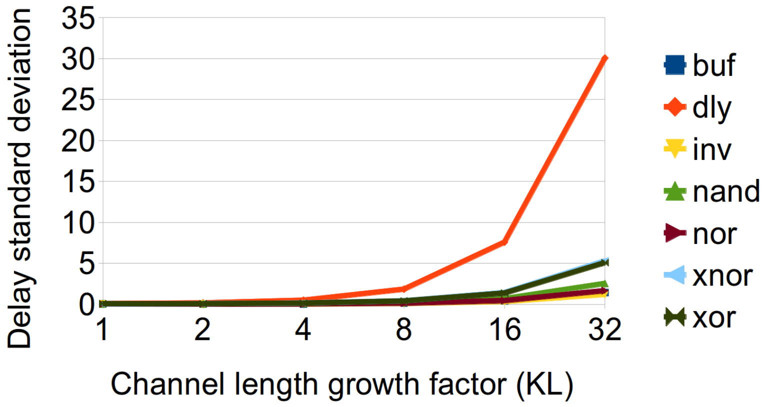


Figure 4.6: Gate delay standard deviation σ_gs a function of K_L ($K_W = 1$)

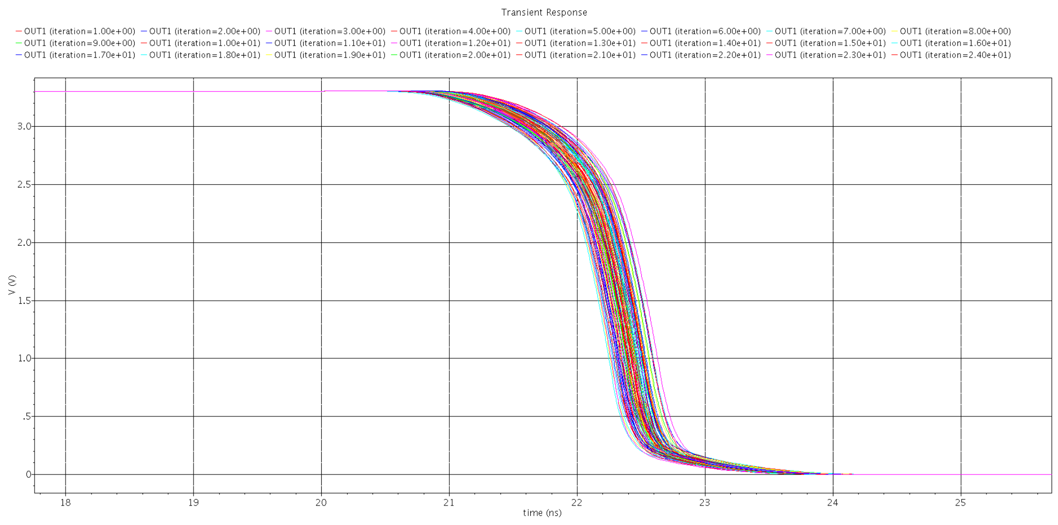


Figure 4.7: Delay variability fast cell

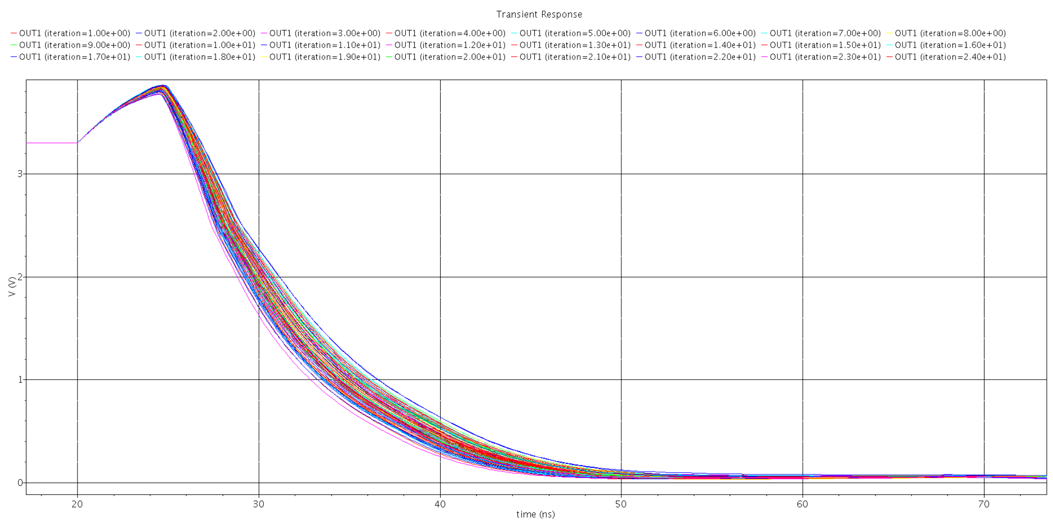


Figure 4.8: Delay variability slow cell

Table 4.2 lists the values of σ_g for all combinational gates in the library and the smallest (1) and the largest (32) values of K_L. Given that APUF design seeks to use gates with the largest delay variability, the best candidate in Table 4.2 is *dly12*.

Gate Type	Mean		Std. Dev. (σ_g)	
	K_L=1	32	K_L=1	32
buf2	0,151	36,460	0,077	1,437
dly12	1,269	**798,758**	0,068	**30,065**
inv0	0,208	20,206	0,052	1,193
nand20	0,299	40,869	0,042	2,540
nor20	0,166	27,709	0,034	1,658
xnor20	0,349	65,242	0,041	5,295
xor20	0.300	87,854	0,059	5,077

Table 4.2: Mean and standard deviation for modified combinational gates

4.3 Arbiter Simulation

The first step a designer must take to design an APUF arbiter is to evaluate all possible storage element candidates according to the the requirements described in Section 3.2. As mentioned before, two possible arbiter candidates are the DFF and the SR-Latch.

As discussed in the first requirement of Section 3.2, a simple analysis of the logic of DFF and SR-Latch shows that the DFF has unbalanced input to output paths, suggesting that the SR-Latch is probably a better choice for an APUF arbiter. On the other hand, a careful analysis of the setup and hold times of the DFF and SR-Latch should be performed to evaluate the storage element which satisfies the second requirement in that section, i.e. the smallest possible forbidden window. This is done to assure that the selected device is the most sensitive one, capable of capturing any small path delay differences produced by the APUF network paths. Section 4.3.1 describes the used procedure and obtained results.

4.3.1 Selecting the arbiter with smallest forbidden window

The *setup time* characterization process for a DFFs and SRLs is well-known [2] and follows a straighfoward procedure which measures the delay from each input to the Q output. For the case of the DFF, two input signals are used: Data (D) and Clock (CLK). In each of the signals it is applied a rising edge step-function out of phase, illustrated by the letters C and D in Figure 4.9.

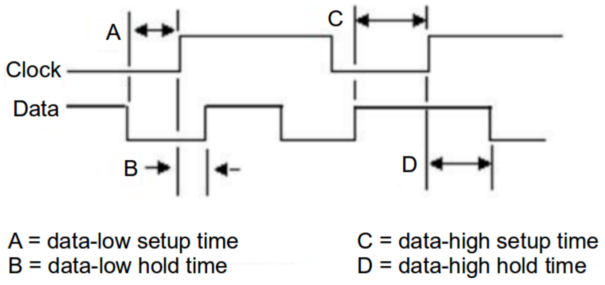


Figure 4.9: Setup and Hold times specification for rising-edge-triggered flip-flop

To determine the *nominal delay*, the experiment is initially configured so that a long time difference is used between the moment that D and CLK transitions are applied. This guarantees that the setup time is satisfied and that the clk-to-Q delay is the fastest possible nominal delay. The next steps in the experiment consists in decreasing the application time between D and CLK and measuring its impact in clock-to-Q. The experiment keeps reducing the difference until the output signal stops transiting to '1'. The setup time (T_{su}) is then defined to be in between 5% to 10% of the nominal delay clk-to-Q (industrial standard). The *hold time* characterization for the DFF is not necessary for this work because, after the transition, both signals stay stable at logic '1' until the next APUF challenge is applied, which just occurs after evaluating the arbiter result. Figure 4.10 illustrates the described procedure and Figure 4.11 shows the circuit used for the DFF arbiter characterization, which also includes a signal to reset the flip-flop to a well-known state at the beginning of every simulation.

For the latch characterization, we defined the equivalent *setup* and *hold time* conditions (Figure 4.12) [2]. The *setup time* is the minimum time that signal S needs to arrive in NAND SR-latch before signal R. In case S arrives very close to R, the output signal stays in a metastable value for a certain time before stabilizing to zero(Figure 4.14). This could take as long as many nanoseconds. The *hold time* is the minimum time that signal R needs to arrive in NAND SR-latch before signal S. In case R arrives very close to S, it causes the output value to bounce to a value that can be lower than V_{ih}[1] or even flip the output (Figure 4.15).

Figure 4.13 shows the circuit used for the SR-Latch arbiter characterization. The experiments have shown that the *setup time* for the SR-Latch can be less strict than the

[1]Minimum input voltage guaranteed to be recognized as a logical "1".

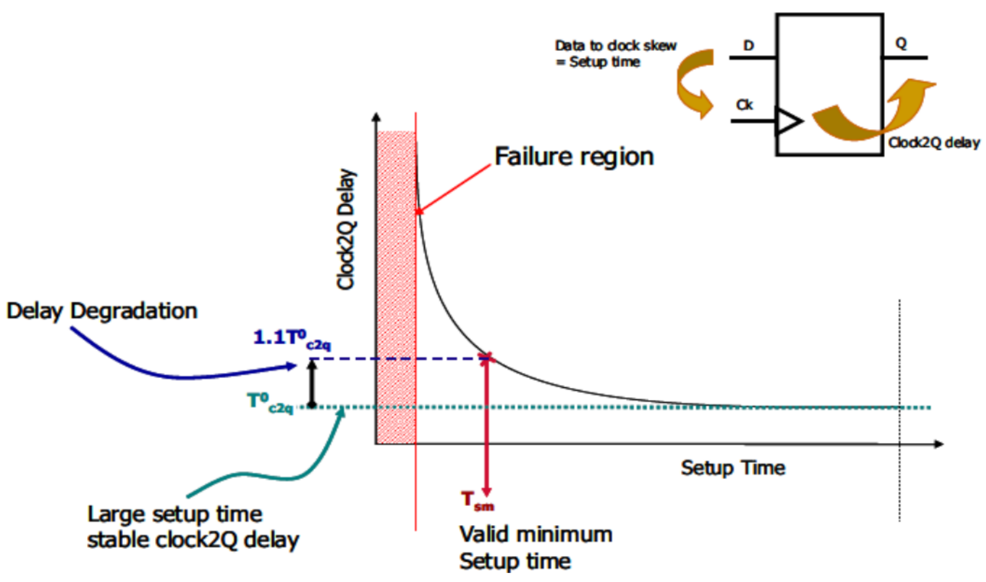


Figure 4.10: DFF characterization procedure [2]

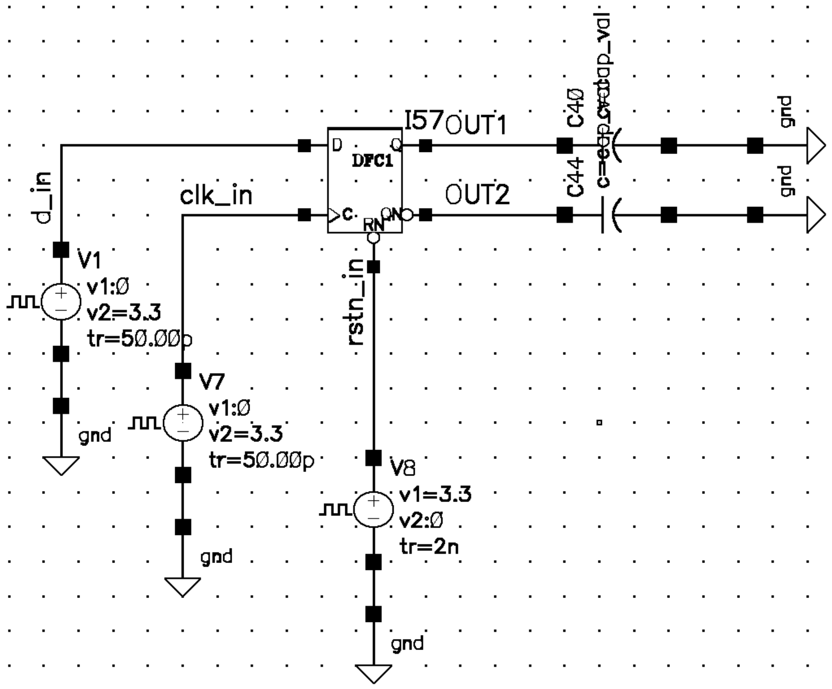
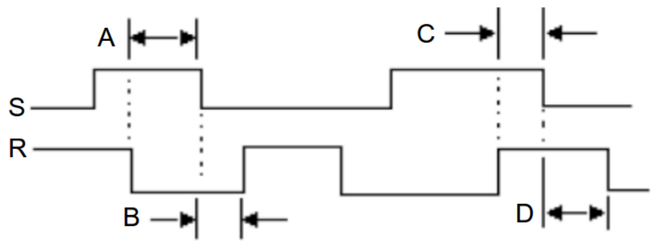


Figure 4.11: DFF arbiter characterization circuit



A = data-low setup time
B = data-low hold time
C = data-high setup time
D = data-high hold time

Figure 4.12: Setup and Hold times specification for SR-Latch

hold time because, if it is available enough time for the circuit to get stable, the registered value is always correct. On opposite, in case of the *hold time* to be violated, the latch can register an incorrect value (flip output).

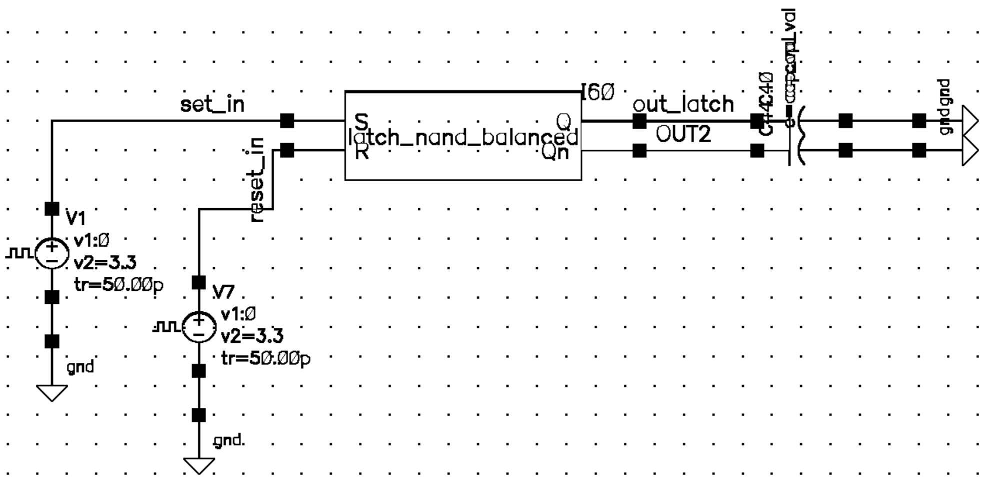


Figure 4.13: SR-Latch arbiter characterization circuit

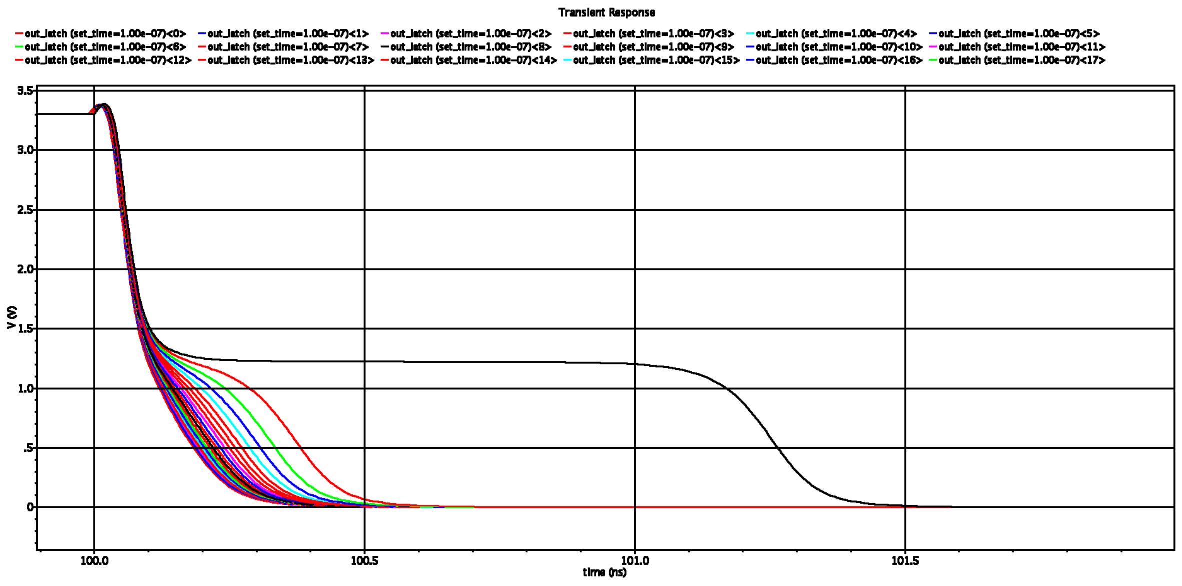


Figure 4.14: SR-Latch arbiter characterization - setup time simulation

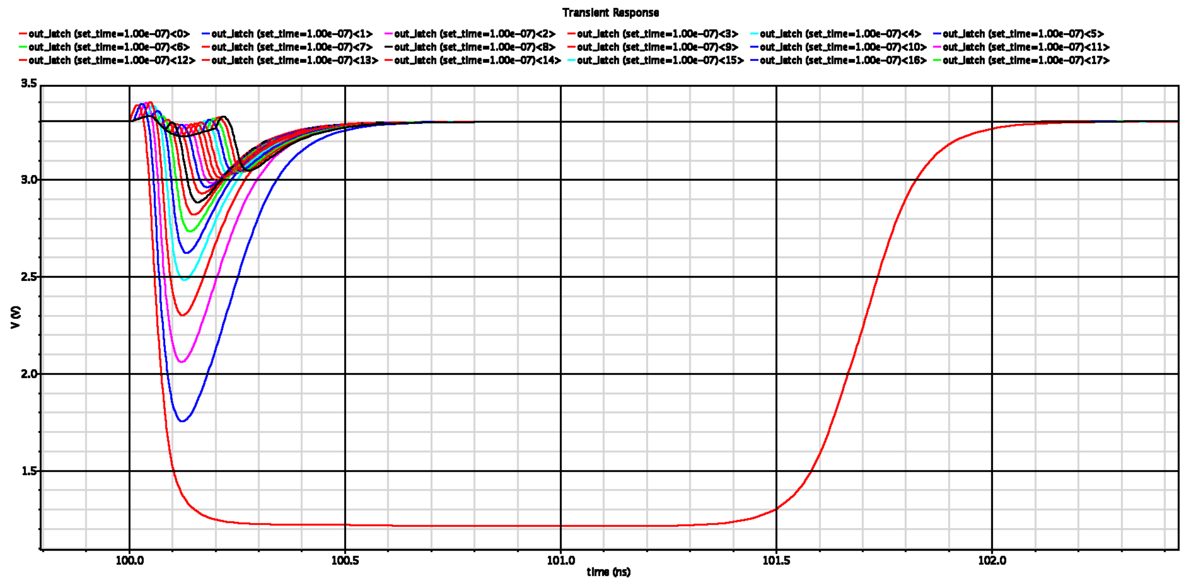


Figure 4.15: SR-Latch arbiter characterization -hold time simulation

Table 4.3 contains the experimental results with the most relevant t_{su} or t_h times for the DFF and SR-Latch built using gates from the library. Based on the presented results, the recommended architecture for the arbiter should be an SR-Latch using NAND28 gates, given that it has the smallest possible t_h.

Arbiter Type	Setup time (DFF) Hold time (SR-Latch)
dfc1	0,1
dfc3	0,08
latch20 balanced	0,00069
latch28 balanced	**0,00052**
latch20 unbalanced	0,03898
latch28 unbalanced	0,032126

Table 4.3: *Setup* or *Hold* times for the DFF and SR-Latch

4.4 Full Circuit Simulation

The goal of this set of experiments is to evaluate what is the impact of the combination of the design techniques discussed in the previous sections, on the overall APUF response variability (i.e. HWD). The following design techniques were evaluated: (a) balanced versus unbalanced arbiters (DFF x Balanced SR-Latch x Unbalanced SR-Latch); (b) high versus low gate strengths (DLY12 x XOR20); and (c) different transistor L sizes ($K_L = 1, 8, 32$). To this purpose, 36 APUF circuit configurations were simulated, as listed in Table 4.4. In the following figures *DFF* stands for D Flip-Flop (Figure 3.3), *SRL* is the balanced SR-Latch (Figure 3.4b) and *USRL* is the unbalanced SR-Latch (Figure 3.4a).

Arbiter		Delay Gate	L scale factor
Type	Cell	Cell	K_L
DFF	dfc1	dly12	1, 8, 32
		xor20	
	dfc3	dly12	
		xor20	
SRL	nand20	dly12	1, 8, 32
		xor20	
	nand28	dly12	
		xor20	
USRL	nand20	dly12	1, 8, 32
		xor20	
	nand28	dly12	
		xor20	

Table 4.4: Full circuit configurations

In order to measure the APUF response quality, the HWD is computed using the methodology proposed in *Katzenbeisser et al* [16] and described in Section 2. Specifically, in this dissertation the same set of N random challenges $C_{ik}, i = 0..N-1, k = 0..M-1$ is applied to each one of the M APUF instances. For a given challenge C_{ik}, the HWD of the output responses R_{ik} is computed across all 32 circuits. By using this configuration, the ideal HWD expected, as described in Section 2.6, is centered around half the number of chips stimulated, which in this case is 16.

Two experiments are performed to evaluate these configurations. In the first one (Section 4.4.2), the goal is to compare the best configuration using DFF and SR-Latch, in order to measure the impact on the HWD when using these two different types of arbiters. In the second experiment (Section 4.4.3), the goal is to evaluate the impact of gate sizing on HWD.

The delay network used in the experiments contains 64 stages, the same number of stages as described by Suh and Devadas in [4]. Unfortunately, although the simulation infrastructure was robust, limitations in the Monte-Carlo analog simulation speed restricted the set of experiments to $M = 32$ APUF circuits and $N = 128$ challenges per circuit. For example, some combinations of challenge-configurations took 12 days to simulate.

4.4.1 Testbench for Full Circuit Simulation

Figure 3.1 shows the circuit used for the full simulation. Figure 4.16 shows three of the 64 stages that compose the delay network. The signals driven in port *IN1* and *IN2* correspond to the step signal, used to create the racing condition between the two paths. The *SEL* ports select the paths to be used as output at each stage. In ports *SEL* it is set the voltage values for the corresponding challenge bits to be applied in the circuit. For example, if the challenge's first three bits are 101, the values in *SEL* ports are: $sel\_0 = 3.3V$, $sel\_1 = 0V$ and $sel\_2 = 3.3V$.

Figure 4.17 contains two of the three arbiter options, used for the full circuit simulation. The top circuit corresponds to the DFF arbiter with reset, and the bottom, the balanced SR-Latch. The unbalanced SR-Latch has the same interface as the former, but its feedback connections are different, as shown in Figure 3.4. To use the desired arbiter, it is only necessary to map the output signals from the delay network (D and C) to the arbiter's inputs. In Figure 4.17, the DFF arbiter is active while the SR-Latch is disabled.

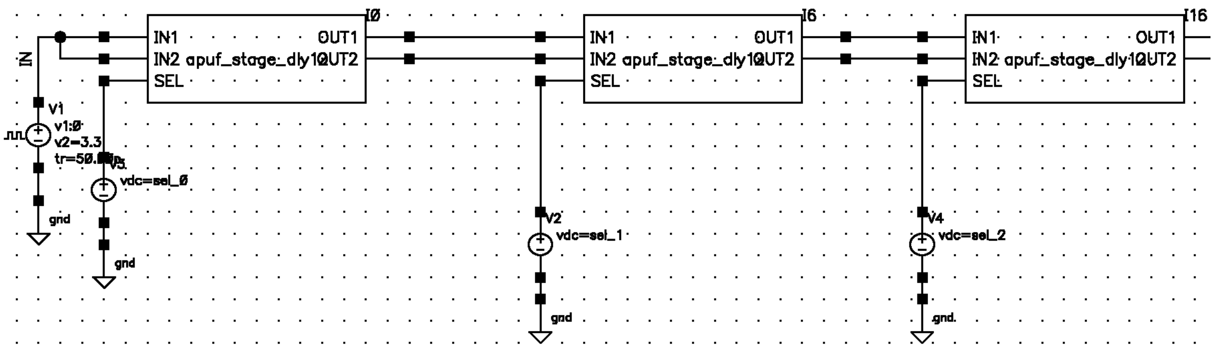


Figure 4.16: Delay network circuit in ADE

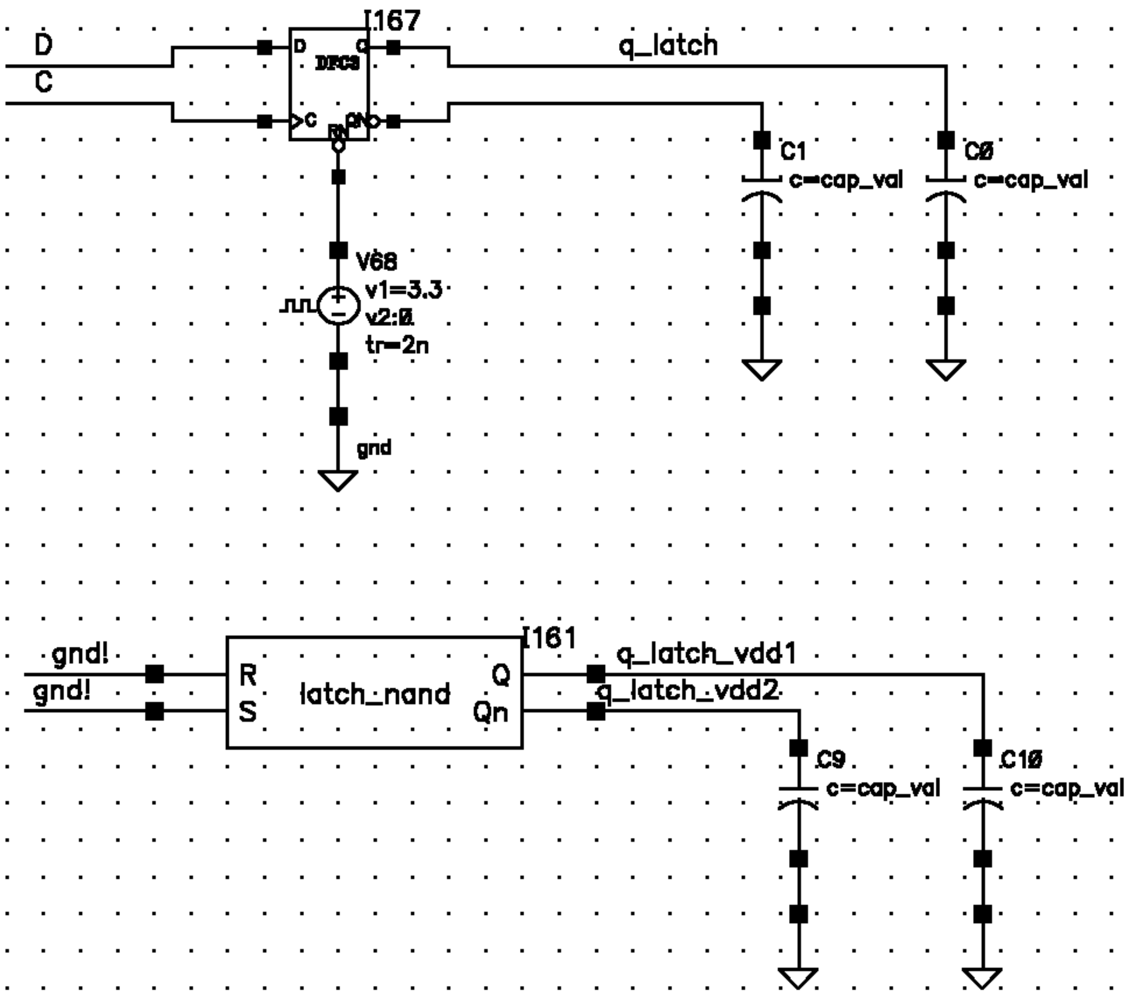


Figure 4.17: Arbiter circuit in ADE

The designed testbench required the usage of SPICE and Monte-Carlo simulation. Its inputs and outputs only support the analog format (voltage, current, etc). For this reason, we created a *simulation framework* to handle the conversions and configurations so that the input challenges and output responses are used in digital format. Figure 4.18 shows such framework.

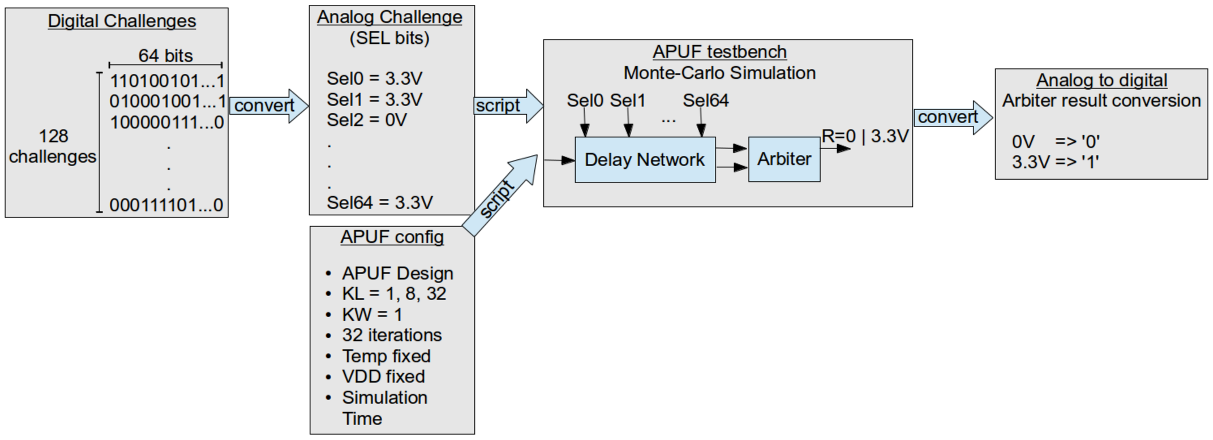


Figure 4.18: Simulation Framework

The framework starts by obtaining the 128 64-bit challenges from the input file, one challenge per line, and for each line it creates a simulation configuration *script* in *OCEAN* (Open Command Environment for Analysis). The script contains the transistor geometry multipliers (K_L and K_W), number of iterations (fixed in 32 circuits), temperature and VDD (fixed), APUF design configuration (Table 4.4) and the values in analog format of each challenge *Sel* bit to be set along the delay network. For every simulation output, the result is converted back to a digital format, which provides an output similar as shown in Listing 4.1. For our case, only the final value next to $500us$ is important, because it is the response obtained after all the internal transitions in the circuit. Each of them correspond to the output result resulting by applying the same challenge to different circuits.

Appendix B describes in more details the procedure for the challenge inputs and circuit simulation.

Listing 4.1: Simulation output file

```
(monte = Task1Monte, iteration = 1)
time (s)          awvAnalog2Digital(VT("q_latch")
                         1.8 0.2 0 0 "hilo") (V)
    0                 St0
   45.4071u           StX
   45.4071u           St1
  500u                St1

# Set No. 2
(monte = Task2Monte, iteration = 2)
time (s)          awvAnalog2Digital(VT("q_latch")
                         1.8 0.2 0 0 "hilo") (V)
0                     St0
500u                  St0

# Set No. 3
(monte = Task3Monte, iteration = 3)
time (s)          awvAnalog2Digital(VT("q_latch")
                         1.8 0.2 0 0 "hilo") (V)
0                     St0
500u                  St0

# Set No. 4
(monte = Task4Monte, iteration = 4)
time (s)          awvAnalog2Digital(VT("q_latch")
                         1.8 0.2 0 0 "hilo") (V)
    0                 St0
   48.8022u           StX
   48.8023u           St1
  500u                St1
```

4.4.2 Comparing HWD for different arbiters

The following experiment analyzes the responses of the full-circuit simulation when using the two best arbiter options with different cells: DFF (DFC3 cell) and balanced SR-Latch (NAND28 gates). The delay network uses DLY12 gates and $K_L = 1$ for both cases. For the comparison, we measured the HWD when 128 64-bit challenges are applied, give that this is the ideal result with distribution centered around 16.

Figure 4.19 shows that the DFF HWD is centered around '6.05' and thus its responses is very biased towards producing more 0s than 1s. On the other hand, the SR-Latch distribution is centered around '15.77', meaning that its responses have almost the same number of 0s and 1s. This experiment reveals that the SR-Latch is a much better arbiter than the DFF, thus validating the hypothesis put forward in Section 3 that the unsymmetrical path delays within the DFF design could adversely affect the APUF response.

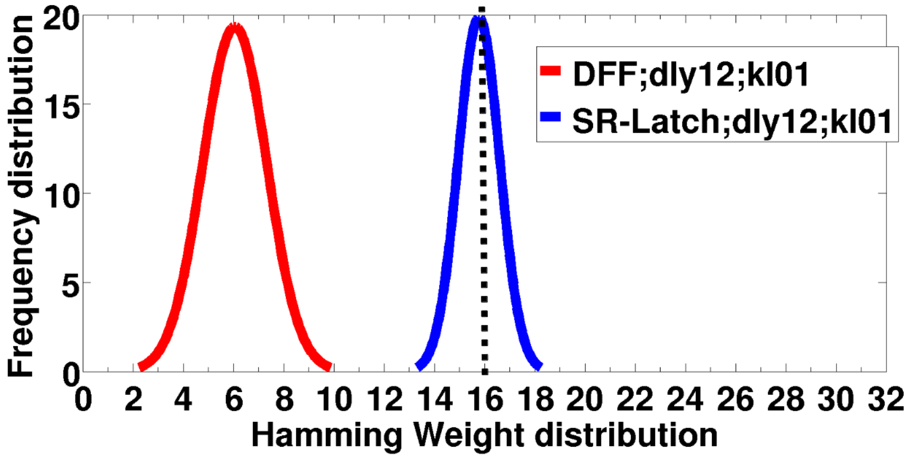


Figure 4.19: HWD comparison when using DFF and SR-Latch as arbiters

4.4.3 Improving HWD using gate sizing

This experiment aims at evaluating the impact of the increase in the gate transistor channel length (L) on the APUF HWD, as raised in Section 3.1.2. Figure 4.20 shows the HWD for a 64-stages APUF, with XOR20 gates in the the delay network and NAND28 gates in the SR-Latch arbiter. HWD is analyzed for $K_L = 1$ and $K_L = 32$ and curve fitting is performed.

Figure 4.20 shows that both HWD exhibit a Gaussian behaviour. However, the APUF with the largest channel length ($K_L = 32$) is centered around '15.51', while the APUF with the smallest channel length ($K_L = 1$) is centered around '14.02', a response slightly biased towards 0. This is an additional confirmation that increasing channel length can contribute to improving the unpredictability of APUFs.

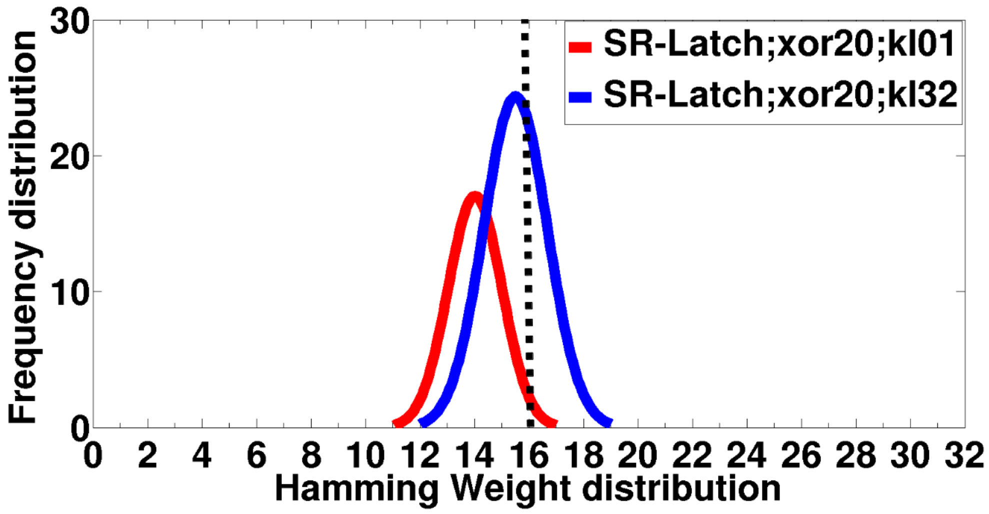


Figure 4.20: HWD comparison when using SR-Latch with different drive strength in delay network

4.4.4 Overall result

Arbiter		delay gate	HW mean			HW std dev		
type	cell		K_L=1	8	32	K_L=1	8	32
DFF	dfc1	dly12	6,11	16,28	16,01	1,27	1,16	1,31
		xor20	0,88	11,05	14,71	0,38	1,11	1,65
	dfc3	dly12	6,05	16,33	15,95	1,26	1,19	1,31
		xor20	1,20	10,98	12,43	0,76	1,22	1,45
SRL	nand20	dly12	15,71	15,11	16,1	0,68	1,64	1,72
		xor20	14,77	15,30	15,43	1,15	0,91	1,07
	nand28	**dly12**	15,77	15,01	**16**	0,81	1,57	**1,79**
		xor20	14,02	15,41	15,51	0,98	1,04	1,16
USRL	nand20	dly12	10,52	14,48	15,95	1,06	1,54	1,75
		xor20	4,39	13,68	14,99	1,21	1,06	1.11
	nand28	dly12	9,59	14,41	15,91	0.87	1.54	1.68
		xor20	1,64	13,33	14,86	0,91	1,14	1,16

Table 4.5: HW Mean and standard deviation for different configurations

Table 4.5 contains the HW mean and standard deviation for all the full circuit simulations performed. The USRL entry of the table represents the result using an unbalanced SR-Latches. As expected, the best configuration for the APUF circuit is obtained with balanced SR-Latch using NAND28 gates, DLY12 as delay cell and $K_L = 32$. On the other hand, the worst performance is obtained with DFF using DFC1 cell, XOR20 as delay gate and $K_L = 1$.

The experimental results show that the latch performs better than the Flip-Flop as arbiter. It also shows the delay networks with larger transistor channel length (L) yields a better Hamming Weight statistical distribution. As a matter of fact, as one increases K_L towards 32, HW mean approaches the ideal value (16), and larger values of HW standard deviation are obtained. On the other hand, the experiments show that using cells with higher drive strength in the SR latch did not have significant impact on overall performance.

Chapter 5

Conclusions and Future Work

Delay-based PUFs are one of the most studied PUFs in the literature and due to its small cost and design simplicity, could eventually be used in low-cost IoT devices. To the best of our knowledge, there is no work that has investigated in this detailed-level, the effect of the structural, electrical and physical characteristics of delay-based PUFs to its response's statistical characteristics. Our work was set out to improve the statistical variability of APUF design by using a collection of techniques usually found on VLSI designs, but not yet applied to PUF designs. It analyzes how the proper selection of arbiter element and gate sizing can affect the delay variability, and how to make the best design decisions based on experimental evaluations.

This work uses extensive Monte Carlo transistor-level SPICE simulation to evaluate how design choices affect the unpredictability of delay-based PUFs. It shows that the combination of the appropriate arbiter and gate size configurations can considerably improve the Hamming Weight distribution of APUF responses. Specifically, it reveals that: (a) the use of *weak* gates along the delay path can improve delay variability by a factor of four; (b) increasing the transistor channel length of gates along the design path by a factor of 32 results in a delay variability one order of magnitude greater; and (c) the use of SR-Latch based arbiters, instead of the widely used DFF arbiter, results in significant improvement in the Hamming Weight distribution of the PUF.

The experiments results show that there is still margin to improve the PUF characteristics focusing on their project parameters.

Another important fact refers to publications that analyze and compare different PUFs architectures (e.g. [16]) without verifying their design parameters variations. Based on our work, this can clearly compromise the comparative results, since the project parameters selection can significantly affect the relative performance of each architecture.

As future work, we intend to test the combination of techniques developed during this research in real silicon implementations, thus allowing the application of a large set of challenges in different kinds of configurations so as to confirm the results obtained by this work.

It is also necessary to analyze the impact of the environmental changes, e.g temperature, voltage variation and aging in the PUF behaviour when the set of techniques developed in this work is applied. As example of this kind of analysis, we need to verify if, at different temperature conditions, the PUF still delivers the same set of responses when the same set of challenges are applied.

We also intend to apply this set of techniques in other delay-based PUFs to verify if we can achieve the same kind of improvements as obtained for the Ozturk's PUF, both in simulation and in real silicon devices.

Bibliography

[1] N.H.E. Weste and D.M. Harris. *CMOS VLSI Design: A Circuits and Systems Perspective*. ADDISON WESLEY Publishing Company Incorporated, 2011.

[2] Savithri Sundareswaran. *Statistical Characterization For Timing Sign-Off: From Silicon to Design and Back to Silicon*. PhD thesis, University of Texas, May 2009.

[3] Gernot Heiling. HIT-Kit 4.0 Training - Introduction to Analog Tutorial, June 2012.

[4] G. Edward Suh and Srinivas Devadas. Physical Unclonable Functions for Device Authentication and Secret Key Generation. DAC '07, New York, NY, USA, 2007. ACM.

[5] Jae W. Lee, Daihyun Lim, Blaise Gassend, G. Edward Suh, Marten van Dijk, and Srinivas Devadas. A Technique to Build a Secret Key in Integrated Circuits for Identification and Authentication Application. In *Proceedings of the Symposium on VLSI Circuits*, 2004.

[6] Blaise Gassend, Dwaine Clarke, Marten van Dijk, and Srinivas Devadas. Silicon Physical Random Functions. In *ACM Conference on Computer and Communications Security*, New York, NY, USA, 2002. ACM Press.

[7] Roel Maes and Ingrid Verbauwhede. Physically Unclonable Functions: A Study on the State of the Art and Future Research Directions. In Ahmad-Reza Sadeghi and David Naccache, editors, *Towards Hardware-Intrinsic Security*, Information Security and Cryptography. Springer Berlin Heidelberg, 2010.

[8] Lang Lin, Dan Holcomb, Dilip Kumar Krishnappa, Prasad Shabadi, and Wayne Burleson. Design Optimization and Security Validation of Sub-Threshold PUFs. In *SECSI-2010*, 2010.

[9] Ryan Helinski, Dhruva Acharyya, and Jim Plusquellic. A Physical Unclonable Function Defined Using Power Distribution System Equivalent Resistance Variations. DAC '09, New York, NY, USA, 2009. ACM.

[10] Pim Tuyls, Geert-Jan Schrijen, Boris Škorić, Jan van Geloven, Nynke Verhaegh, and Rob Wolters. Read-Proof Hardware from Protective Coatings. CHES'06, Berlin, Heidelberg, 2006. Springer-Verlag.

[11] Blaise Gassend. Physical Random Functions. Master's thesis, Massachusetts Institute of Technology, January 2003.

[12] E. Ozturk, G. Hammouri, and B. Sunar. Physical unclonable function with tristate buffers. ISCAS 2008.

[13] Mehrdad Majzoobi, Farinaz Koushanfar, and Miodrag Potkonjak. Testing Techniques for Hardware Security. In *Test Conference, 2008. ITC 2008. IEEE International*, 2008.

[14] Mehrdad Majzoobi, Farinaz Koushanfar, and Miodrag Potkonjak. Techniques for Design and Implementation of Secure Reconfigurable PUFs. *ACM Trans. Reconfigurable Technol. Syst.*, 2(1), 2009.

[15] Ulrich Rührmair, Frank Sehnke, Jan Sölter, Gideon Dror, Srinivas Devadas, and Jürgen Schmidhuber. Modeling Attacks on Physical Unclonable Functions. In *Proceedings of the 17th ACM Conference on Computer and Communications Security*, CCS '10, pages 237–249, New York, NY, USA, 2010. ACM.

[16] Stefan Katzenbeisser, Ünal Kocabaş, Vladimir Rožić, Ahmad-Reza Sadeghi, Ingrid Verbauwhede, and Christian Wachsmann. PUFs: Myth, Fact or Busted? A Security Evaluation of Physically Unclonable Functions (PUFs) Cast in Silicon. CHES 2012. 2012.

[17] Leonid Bolotnyy and Gabriel Robins. Physically Unclonable Function-Based Security and Privacy in RFID Systems. PERCOM '07, Washington, DC, USA, 2007. IEEE Computer Society.

[18] Ravikanth S. Pappu, Ben Recht, Jason Taylor, and Niel Gershenfeld. Physical One-Way Functions. *Science*, 297, 2002.

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

[20] S. Stanzione and G. Iannaccone. Silicon Physical Unclonable Function resistant to a 1025-trial brute force attack in 90 nm CMOS. In *VLSI Circuits, 2009 Symposium on*, pages 116–117, June 2009.

[21] Blaise Gassend, Marten Van Dijk, Dwaine Clarke, Emina Torlak, Srinivas Devadas, and Pim Tuyls. Controlled Physical Random Functions and Applications. *ACM Trans. Inf. Syst. Secur.*, 10(4):3:1–3:22, January 2008.

[22] Duane S. Boning and Sani Nassif. Models of Process Variations in Device and Interconnect. In *Design of High Performance Microprocessor Circuits, chapter 6*. IEEE Press, 1999.

[23] Sergey Morozov, Abhranil Maiti, and Patrick Schaumont. A comparative analysis of delay based puf implementations on fpga. *IACR ePrint tbd/2009 (submitted December 19, 2009)*, 2009.

[24] M. Majzoobi, F. Koushanfar, and S. Devadas. FPGA PUF using programmable delay lines. In *Information Forensics and Security (WIFS), 2010 IEEE International Workshop on*, pages 1–6, Dec 2010.

[25] J.H. Anderson. A PUF design for secure FPGA-based embedded systems. In *Design Automation Conference (ASP-DAC), 2010 15th Asia and South Pacific*, pages 1–6, Jan 2010.

[26] Kedar Patel. *Intrinsic and Systematic Variability in Nanometer CMOS Technologies*. PhD thesis, EECS Department, University of California, Berkeley, Dec 2010.

[27] Shahin Tajik, Enrico Dietz, Sven Frohmann, Jean-Pierre Seifert, Dmitry Nedospasov, Clemens Helfmeier, Christian Boit, and Helmar Dittrich. Physical Characterization of Arbiter PUFs. CHES 2014, 2014.

[28] Daisuke Suzuki and Koichi Shimizu. The Glitch PUF: A New Delay-PUF Architecture Exploiting Glitch Shapes. CHES 2010. Springer Berlin / Heidelberg, 2011.

[29] Wei Zhao, Yu Cao, F. Liu, K. Agarwal, D. Acharyya, S. Nassif, and K. Nowka. Rigorous extraction of process variations for 65nm CMOS design. In *Solid State Device Research Conference, 2007. ESSDERC 2007. 37th European*, pages 89–92, Sept 2007.

[30] AMS AG. 0.35μm *CMOS Digital Standard Cell Databook*, 2012.

[31] Lang Lin, Dan Holcomb, Dilip Kumar Krishnappa, Prasad Shabadi, and Wayne Burleson. Low-power sub-threshold design of secure physical unclonable functions. In *Low-Power Electronics and Design (ISLPED), 2010 ACM/IEEE International Symposium on.*

Appendix A

Tutorial: Monte-Carlo Simulation

In this chapter it is explained how to setup the environment and perform the Monte-Carlo simulations described in the previous sessions. The objective is to provide the required knowledge for the ones interested in reproduce the results using the same tools as used in this work.

A.1 Environment Setup

In order to configure the environment variables in a linux terminal, the script showed in Script A.1 should be present in the user home folder. As example, this script is named as "setup.cadence". In order to call the script in every instance of the terminal, the line below should be added in the ".bashrc" file.

```
source ~/setup.cadence
```

If the setup was performed correctly, the output of the command "echo $CDSHOME" should return the equivalent result: "\tools\cadence\IC5141". To start the design and simulation tool, one can simply run the AMS script:

```
>>ams_cds -mode icfb
```

If asked, the technology "C35B4" should be associated to the project.

Listing A.1: Environment setup script

```
#!/bin/bash

export LM_LICENSE_FILE=5280@v440:26745@medalha

export CADENCE_HOME=/tools/cadence

export CDS_LIC_TIMEOUT=10

export MMSIM_HOME=${CADENCE_HOME}/MMSIM101/tools/bin
export ETSHOME=${CADENCE_HOME}/ETS110
export CDS_ROOT=${CADENCE_HOME}/IC5141
export CDSHOME=${CADENCE_HOME}/IC5141
export CDSDIR=${CADENCE_HOME}/IC5141
export EDIHOME=${CADENCE_HOME}/EDI101
export ASSURA_AUTO_64BIT=NONE
export ASSURAHOME=${CADENCE_HOME}/ASSURA/5141USR2
export LANG=C
export MOZILLA_HOME="/usr/bin/firefox"
#export CDS_AUTO_64BIT ALL
export CDS_BIND_TMP_DD=both
export VRST_HOME=${CADENCE_HOME}/INCISIV102
export SPECMAN_WEB_BROWSER=/usr/bin/firefox
export RC_HOME=${CADENCE_HOME}/RC111

export AMS_DIR=/tools/techlib/hk380

export PATH="$CDSHOME/share/bin:$PATH"
export PATH="$CDSHOME/bin:$PATH"
export PATH="$CDSHOME/tools/bin:$PATH"
export PATH="$CDSHOME/tools/dfII/bin:$PATH"

export PATH="$EDIHOME/share/bin:$PATH"
export PATH="$EDIHOME/bin:$PATH"
export PATH="$EDIHOME/tools/bin:$PATH"
export PATH="$EDIHOME/tools/dfII/bin:$PATH"

export PATH="$ETSHOME/share/bin:$PATH"
export PATH="$ETSHOME/bin:$PATH"
export PATH="$ETSHOME/tools/bin:$PATH"
export PATH="$ETSHOME/tools/dfII/bin:$PATH"

export PATH="$VRST_HOME/bin:$PATH"
export PATH="$VRST_HOME/tools/bin:$PATH"
export PATH="$VRST_HOME/share/bin:$PATH"

export PATH="$RC_HOME/bin:$PATH"
export PATH="$RC_HOME/tools/bin:$PATH"
export PATH="$RC_HOME/share/bin:$PATH"

export PATH="$MMSIM_HOME/../../share/bin:$PATH"
export PATH="$MMSIM_HOME/../bin:$PATH"
export PATH="$MMSIM_HOME/../../bin:$PATH"

export PATH="$AMS_DIR/artist/bin:$PATH"

export LD_LIBRARY_PATH
export LD_LIBRARY_PATH="$CDSHOME/tools/lib:$LD_LIBRARY_PATH"
export LD_LIBRARY_PATH="$EDIHOME/tools/lib:$LD_LIBRARY_PATH"
export LD_LIBRARY_PATH="$ETSHOME/tools/lib:$LD_LIBRARY_PATH"
export LD_LIBRARY_PATH="$MMSIM_HOME/../lib:$LD_LIBRARY_PATH"
export LD_LIBRARY_PATH="$VRST_HOME/tools/
                                lib:$LD_LIBRARY_PATH"
export LD_LIBRARY_PATH="$RC_HOME/tools/lib:$LD_LIBRARY_PATH"
export LBS_CLUSTER_MASTER=$(uname -n | sed 's/\(\..*\)*//g')
export CDS_QUEUE_CONF_FILE=/tools/cadence/queue_config
```

A.2 Monte-Carlo Simulation of a 'Inverter' Gate

When the tool starts, several windows pop-ups on the screen. The two entitled "What's new" windows should be closed. At this time, one should be seeing two windows, named "icms" and "Library Manager". The icms (*integrated circuit mixed signal*) is the main window, and from there, one can open the other tools, including the Library Manager. The Library Manager handles all the project being developed, and also loads the standard cells and basics components provided by the foundry.

A.2.1 Creating new library and cellview

To create a new project, go to the *Library Manager* and selects *File* → *New* → *Library*. Figure A.22 shows the opened window, which the library name is added, in this case *MCSIM_INV_GATE*. Select the prefered directory and press *OK*. After, the window showed in Figure A.2 asks for the associated technology for the library. Select "*Attach to an existing techfile*" and on window showed in Figure A.3 select "*TECH_C35B4*".

To create a cellview, go to the *Library Manager*, selects the created library (*MCSIM_INV_GATE*), and then selects on menu *File* → *New* → *CellView*. Figure A.4 shows the opened window. The cellview name on this example is "*inv_gate*".

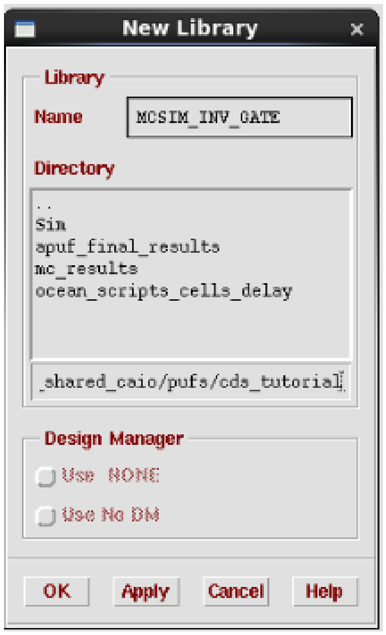
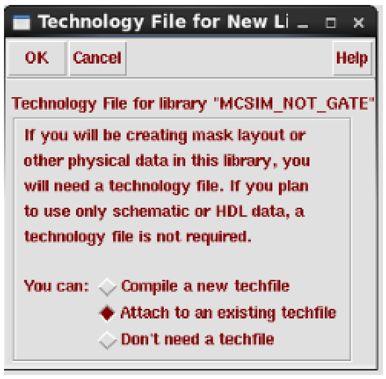


Figure A.1: Adding new library

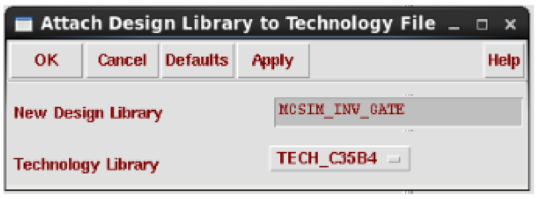


Technology File for library "MCSIM_NOT_GATE"

If you will be creating mask layout or other physical data in this library, you will need a technology file. If you plan to use only schematic or HDL data, a technology file is not required.

You can: Compile a new techfile [✓ Attach to an existing techfile] Don't need a techfile

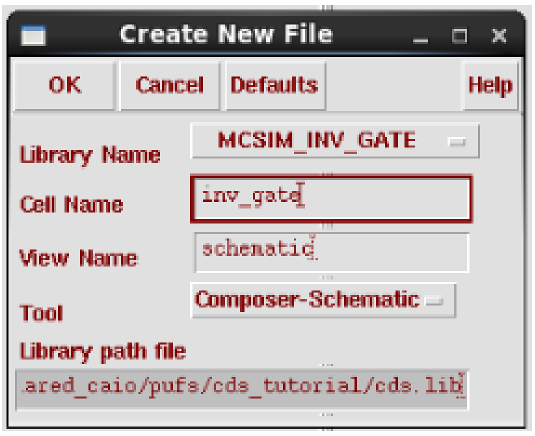
Figure A.2: Technology file for new library



New Design Library: MCSIM_INV_GATE

Technology Library: TECH_C35B4

Figure A.3: Attach design library to technology file



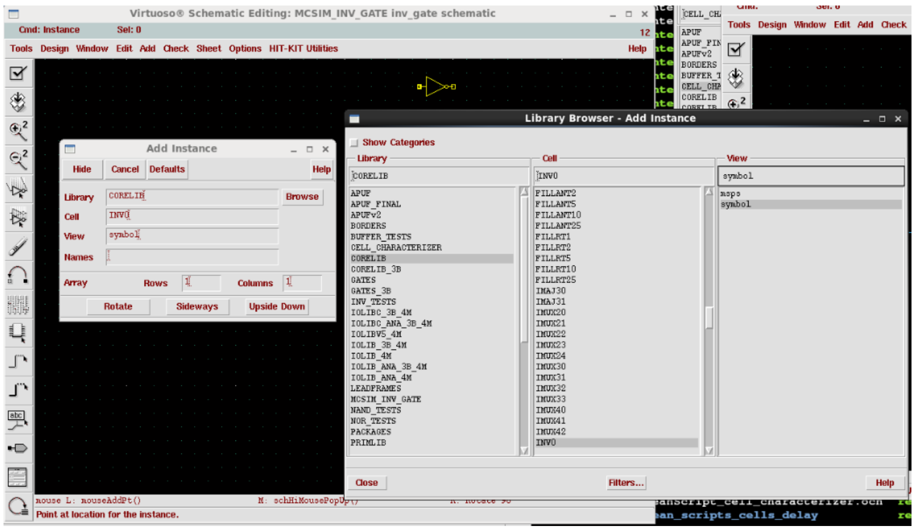
Library Name: MCSIM_INV_GATE
Cell Name: inv_gate
View Name: schematic
Tool: Composer-Schematic
Library path file
ared_caio/pufs/cds_tutorial/cds.lib

Figure A.4: Create new cellview

A.2.2 Circuit Design

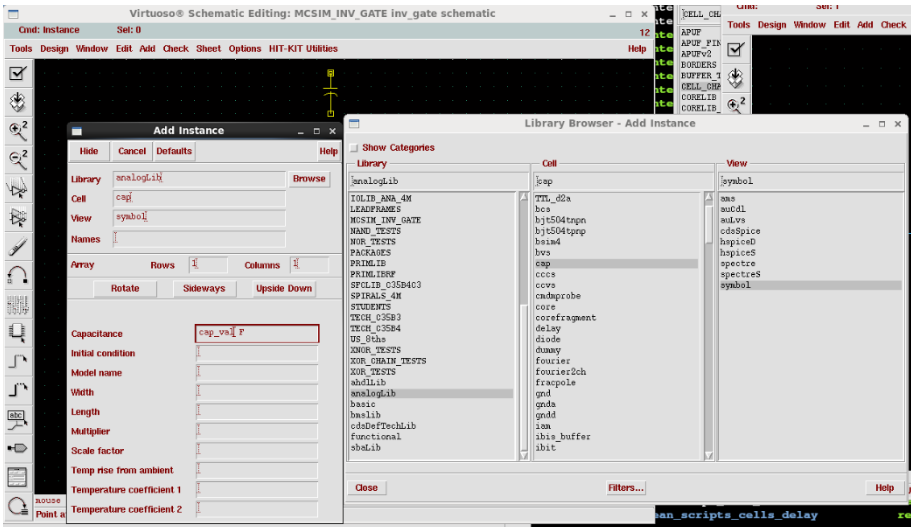
The designed circuit showed in this chapter were used in the experiments described in Section 3.1, to rank the available cells according to their delay variability. The circuit is composed of three cells of the same type, connected in a chain, in which the middle cell is analyzed. This technique makes the input rising edge and the output load being simulated in real conditions. From the schematic window, to add a new instance, as showed in Figure A.5, one can click on button "*instance*" on the left bar, or press "*i*". On library, select "*CORELIB*", cell "*INV0*", and view "*symbol*", than just click on an empty space of the *schematic* window to add the instance. To add an capacitor, one should perform the same procedure but use the library "*analogLib*", cell "*cap*", and on "*Capacitance*" field, put "*cap_ val*". The aforementioned parameter is a variable that will be defined on the next steps. It is also necessary to add the input and output pins for the circuit, as showed in Figure A.7. Finally, to connect all the instances, select the "*wire (narrow)*" button, or press "*w*", to select the wire tool. Connect all the elements to have a final circuit as showed in Figure A.8. To verify if the circuit is correct, and also save it, click in "*Check and Save*" or press "*x*".

From the designed circuit, it is necessary to create the *symbol* view, so that the designed cell can be further added in the testbench to be evaluated. For that, select *Design* → *CreateCellview* → *FromCellview*, as showed in Figure A.9, and press OK.


Virtuoso® Schematic Editing: MCSIM_INV_GATE inv_gate schematic
Cmd: Instance
Sel: 0
Tools Design Window Edit Add Check Sheet Options HIT-KIT Utilities Help
Add Instance
Hide Cancel Defaults Help
Library Browse
Cell
View
Names
Array Rows Columns
Rotate Sideways Upside Down
Library Browser - Add Instance
Show Categories
Library
Cell
View
Close Filters... Help
mouse L: mouseAddPt()
Point at location for the instance.

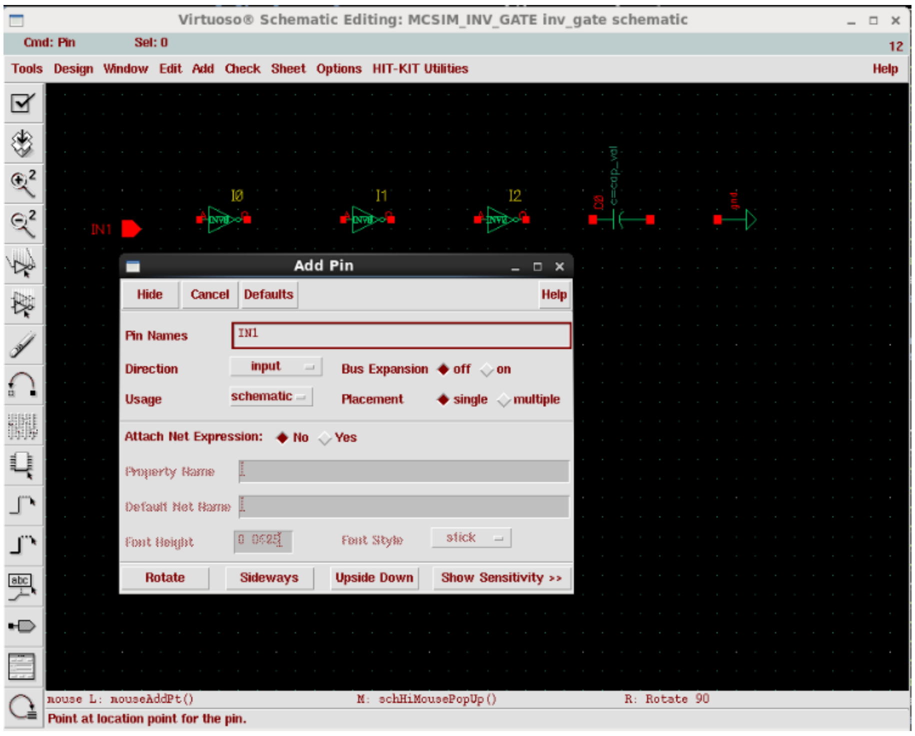
Library: CORELIB
Cell: INV0
View: symbol
Rows: 1
Columns: 1

Figure A.5: Adding a new instance of inverter gate


Virtuoso® Schematic Editing: MCSIM_INV_GATE inv_gate schematic
Cmd: Instance
Sel: 0
Tools Design Window Edit Add Check Sheet Options HIT-KIT Utilities Help
Add Instance
Hide Cancel Defaults Help
Library Browse
Cell
View
Names
Array Rows Columns
Rotate Sideways Upside Down
Library Browser - Add Instance
Show Categories
Library
Cell
View
Close Filters... Help

Library: analogLib
Cell: cap
View: symbol
Rows: 1
Columns: 1
Capacitance: cap_val F
Initial condition
Model name
Width
Length
Multiplier
Scale factor
Temp rise from ambient
Temperature coefficient 1
Temperature coefficient 2

Figure A.6: Adding a new instance of capacitor



Pin Names IN1

Direction input

Bus Expansion [✓ off] on

Usage schematic

Placement [✓ single] multiple

Attach Net Expression: [✓ No] Yes

Property Name

Default Net Name

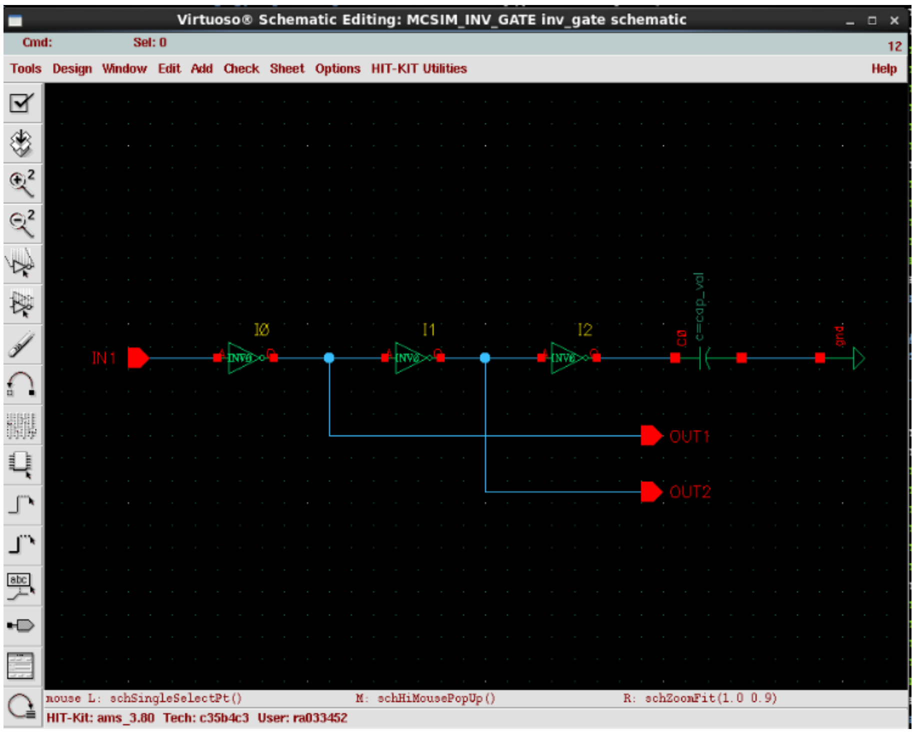
Font Height 0.0625

Font Style stick

mouse L: mouseAddPt() M: schHiMousePopUp() R: Rotate 90

Point at location point for the pin.

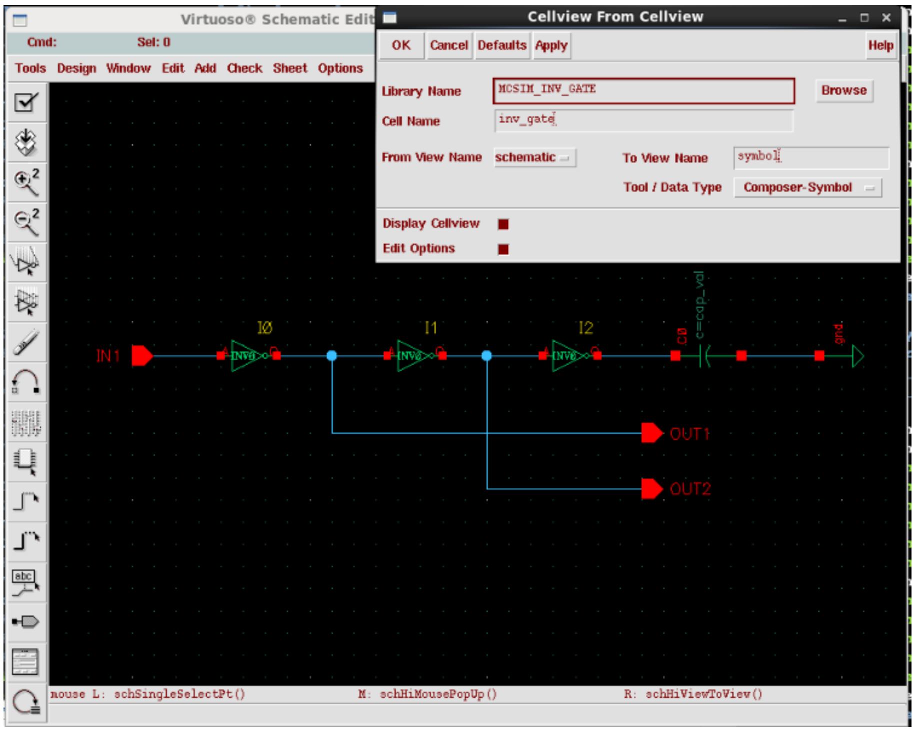
Figure A.7: Add pin



mouse L: schSingleSelectPt() M: schHiMousePopUp() R: schZoomFit(1.0 0.9)

HIT-Kit: ams_3.80 Tech: c35b4c3 User: ra033452

Figure A.8: Inverter circuit ready



Cellview From Cellview

Library Name: MCSIM_INV_GATE
Cell Name: inv_gate
From View Name: schematic
To View Name: symbol
Tool / Data Type: Composer-Symbol
Display Cellview ■
Edit Options ■

mouse L: schSingleSelectPt() M: schHiMousePopUp() R: schHiViewToView()

Figure A.9: Create symbol from schematic

A.2.3 Testbench Design

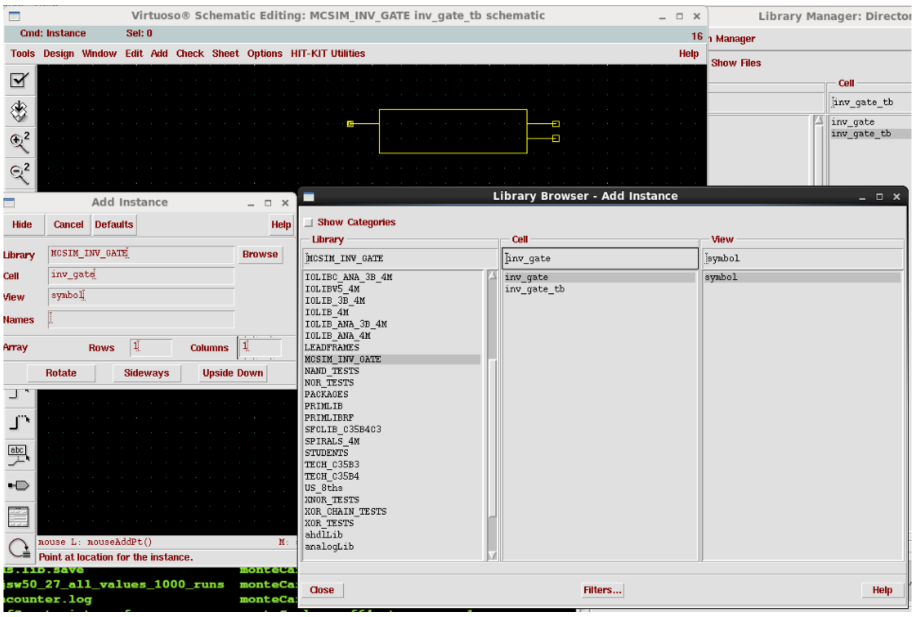
To create the testbench for the circuit previously designed, one should create a new cellview, in this case named "*inv_gate_tb*". To add the "*inv_gate*" instance, click on "*instance*" and select, as showed in Figure A.10, library "*MCSIM_INV_GATE*", cell "*inv_gate*", and view "*symbol*", than just click on an empty space of the *schematic* window to add the instance. After, it is necessary to add the source voltage and the signal input of the circuit. First, select from the *analogLib* the instance "*vdd*" and "*gnd*", than add to the project the instance "*vdc*". For the DC voltage field, add variable name "*vdd_val*", as showed in Figure A.11.

To stimulate the circuit, it is used a rising edge signal, which is generated by the "*vpulse*" instance, also available in *analogLib*. The configuration of this instance is present in Figure A.12. Finally, it is necessary to add a name in the nets to be plotted during the simulation. As showed in Figure A.13, one should click on "*wire name*", and fill the "*Names*" with the wire name, in this case named "*IN*". Then, click on the wire that should receive the name.

From the previous explanation, perform the next steps in order to have the full testbench circuit as showed in Figure A.14. To verify if the circuit is correct, and also save it, click in "*Check and Save*" or press "*x*".

A.2.4 Running the simulation

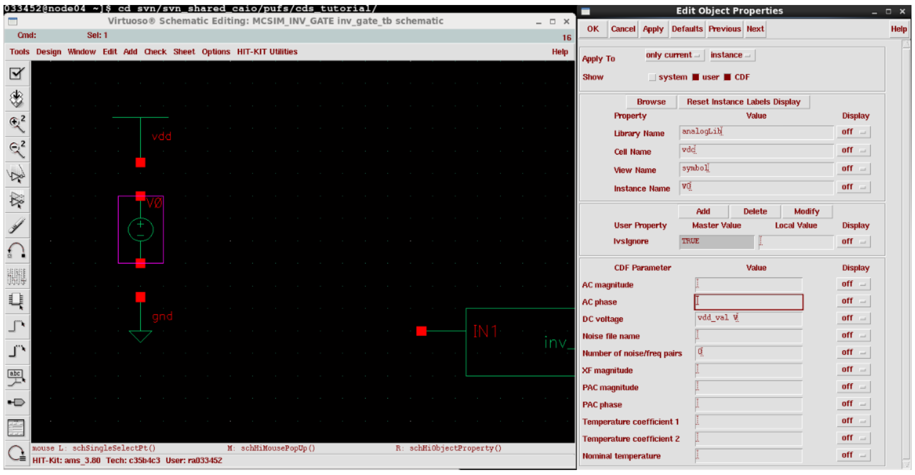
At this moment, all the required circuits are ready to be simulated. From the testbench schematic window, select *Tools*$\rightarrow$ *Analog Environment* to open the simulation environment. After, it is necessary to configure the used libraries for the simulation. As default,



Library	MCSIM_INV_GATE
Cell	inv_gate
View	symbol
Names	
Rows	1
Columns	1

```
mouse L: mouseAddPt()
Point at location for the instance.
```

Figure A.10: Add new instance of inverter circuit



```
033452@node04 ~]$ cd svn/svn_shared_caio/pufs/cds_tutorial/
```

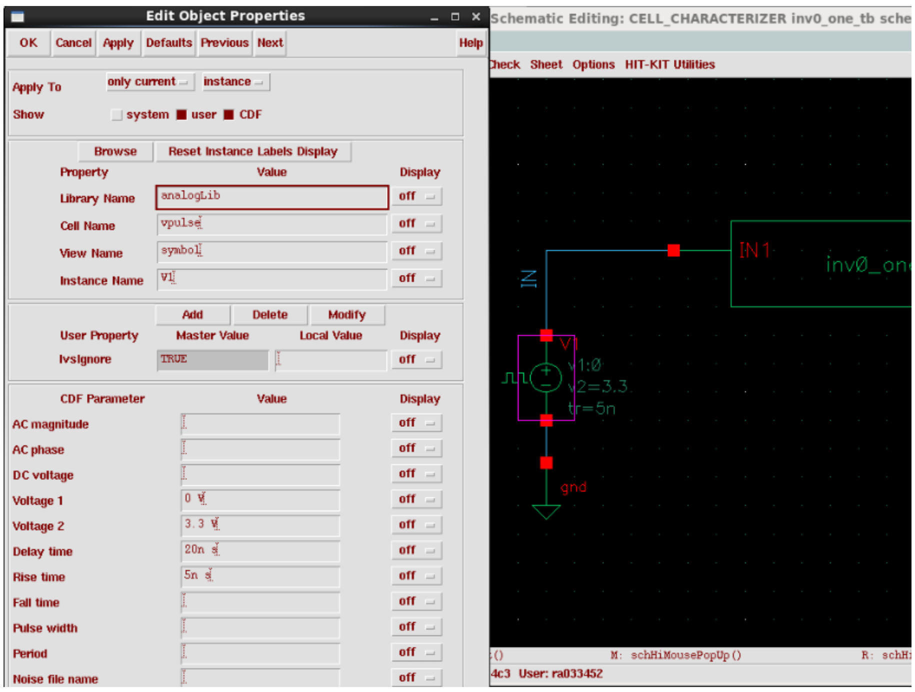
```
mouse L: schSingleSelectPt()        M: schHiMousePopUp()        R: schHiObjectProperty()
HIT-Kit: ams_3.80  Tech: c35b4c3  User: ra033452
```

Property	Value	Display
Library Name	analogLib	off
Cell Name	vdc	off
View Name	symbol	off
Instance Name	V0	off

User Property	Master Value	Local Value	Display
lvsIgnore	TRUE		off

CDF Parameter	Value	Display
AC magnitude		off
AC phase		off
DC voltage	vdd_val V	off
Noise file name		off
Number of noise/freq pairs	0	off
XF magnitude		off
PAC magnitude		off
PAC phase		off
Temperature coefficient 1		off
Temperature coefficient 2		off
Nominal temperature		off

Figure A.11: Add vdd source


Edit Object Properties
OK Cancel Apply Defaults Previous Next Help
Schematic Editing: CELL_CHARACTERIZER inv0_one_tb sche
heck Sheet Options HIT-KIT Utilities
IN
IN1
inv0_one
V1
v1:0
v2=3.3
tr=5n
gnd
M: schHiMousePopUp()
R: schHi
4c3 User: ra033452

Apply To: only current, instance

Show: system, user, CDF

Browse | Reset Instance Labels Display

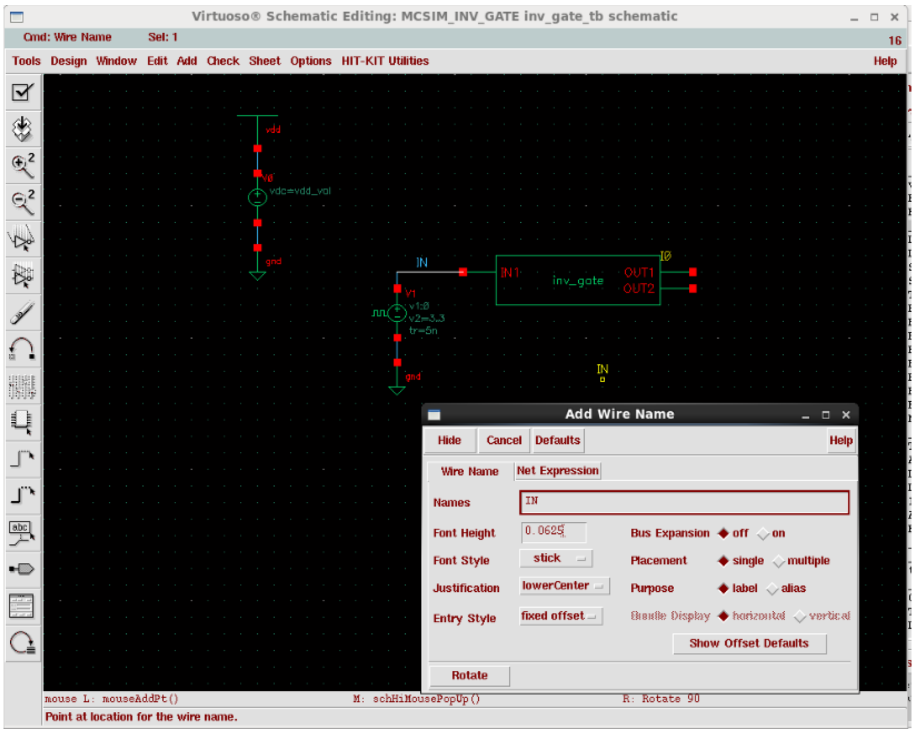
Property	Value	Display
Library Name	analogLib	off
Cell Name	vpulse	off
View Name	symbol	off
Instance Name	V1	off

Add | Delete | Modify

User Property	Master Value	Local Value	Display
lvsIgnore	TRUE		off

CDF Parameter	Value	Display
AC magnitude		off
AC phase		off
DC voltage		off
Voltage 1	0 V	off
Voltage 2	3.3 V	off
Delay time	20n s	off
Rise time	5n s	off
Fall time		off
Pulse width		off
Period		off
Noise file name		off

Figure A.12: Add vpulse source



Wire Name | Net Expression

Names: IN

Font Height: 0.0625

Font Style: stick

Justification: lowerCenter

Entry Style: fixed offset

Bus Expansion: [✓ off] on

Placement: [✓ single] multiple

Purpose: [✓ label] alias

Bundle Display: [✓ horizontal] vertical

Point at location for the wire name.

Figure A.13: Add wire name

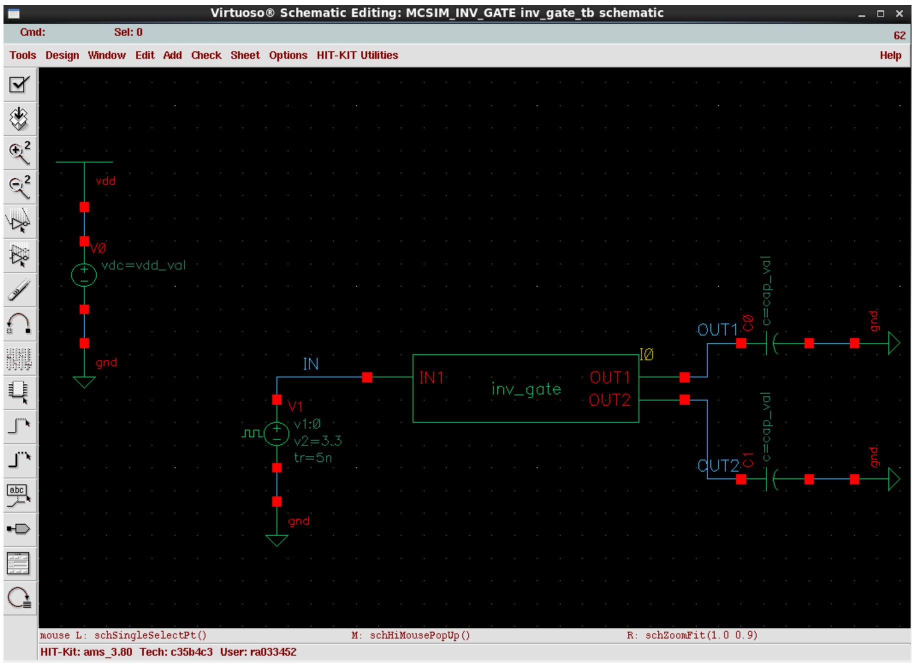

Virtuoso® Schematic Editing: MCSIM_INV_GATE inv_gate_tb schematic
Cmd:
Sel: 0
62
Tools Design Window Edit Add Check Sheet Options HIT-KIT Utilities Help
vdd
V0
vdc=vdd_val
gnd
IN
V1
v1:0
v2=3.3
tr=5n
gnd
IN1
inv_gate
OUT1
OUT2
I0
OUT1
OUT2
C0
C1
c=cap_val
c=cap_val
gnd
gnd
mouse L: schSingleSelectPt()
M: schHiMousePopUp()
R: schZoomFit(1.0 0.9)
HIT-Kit: ams_3.80 Tech: c35b4c3 User: ra033452

Figure A.14: Testbench circuit ready

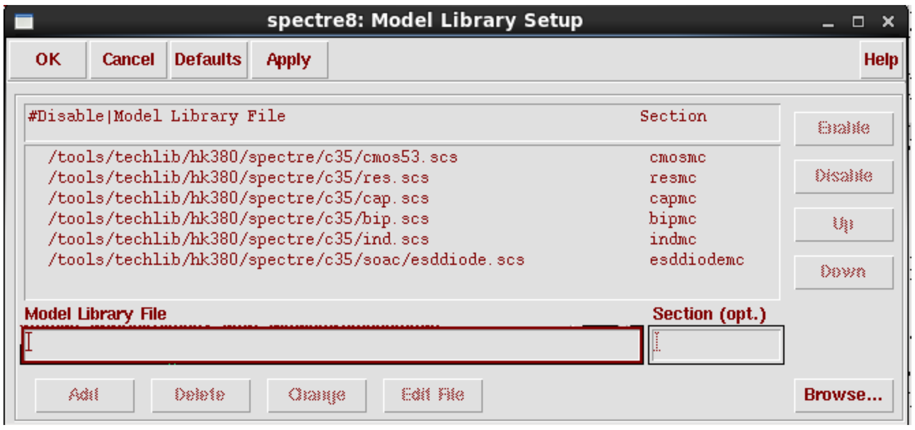
the loaded libraries are used to run the circuit in typical conditions. To change it to get the Monte-Carlo parameters, in the *Analog Environment* window, select *Setup*→ *Model Libraries*. In each of the loaded libraries, change the "*Section*" from *tm to *mc, e.g., *cmostm* to *cmosmc*, as showed in Figure A.15.

In section Section A.2.2 and Section A.2.3, two variables were defined and in Figure A.16 it is showed how to set their values. To open the "*editing design variables*" window, select *Variables*→ *Edit*. The variables "*vdd_val*" should be set to "*3.3*" and "*cap_val*" to "*0.01f*". The units of each variable is automatically defined by the tool.

It is necessary to configure which ports or nets will be evaluated and plotted in the result graph. For that, select *Outputs*→ *Setup* and configure the fields as showed in Figure A.17. Note that */OUT1* corresponds to the wire name, and not the *inv_gate* output port.

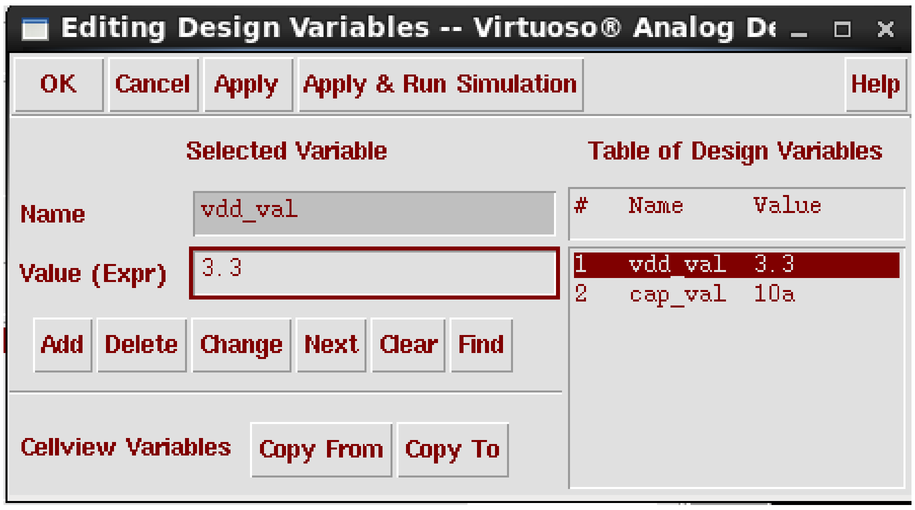
Finnaly, select *Tools*→ *Monte Carlo* to open the window as showed in Figure A.18. In the field "*Number of Runs*" select the number of iterations for the Monte-Carlo simulation. In this work, each iteration is considered a different chip instance. In "*Analysis Variation*" there exist three options: "*Process Only*", "*Mismatch Only*" and "*Process & Mismatch*". The first represents the imperfections that happen due to the manufacturing process and only affect different instances. The second represents the local variations that may happen at the same instance of a chip, causing different transistors to have slightly different behaviours. The third represents the union between the first two models. At the same window, check the box "*Save Data Between Runs to Allow Family Plots*" and then select *Simulation*→ *Run*. After the simulation, the results will be plotted in the graph as showed in Figure A.19.

Simulation -> run



```
#Disable|Model Library File                               Section
 /tools/techlib/hk380/spectre/c35/cmos53.scs               cmosmc
 /tools/techlib/hk380/spectre/c35/res.scs                  resmc
 /tools/techlib/hk380/spectre/c35/cap.scs                  capmc
 /tools/techlib/hk380/spectre/c35/bip.scs                  bipmc
 /tools/techlib/hk380/spectre/c35/ind.scs                  indmc
 /tools/techlib/hk380/spectre/c35/soac/esddiode.scs        esddiodemc
```

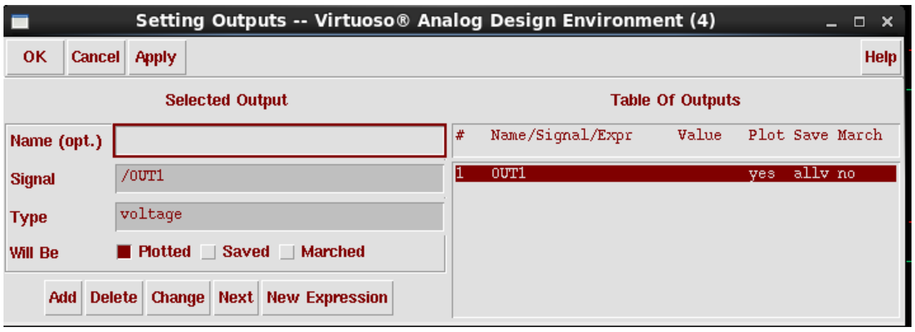
Figure A.15: Setup model libraries



Name: vdd_val
Value (Expr): 3.3

#	Name	Value
1	vdd_val	3.3
2	cap_val	10a

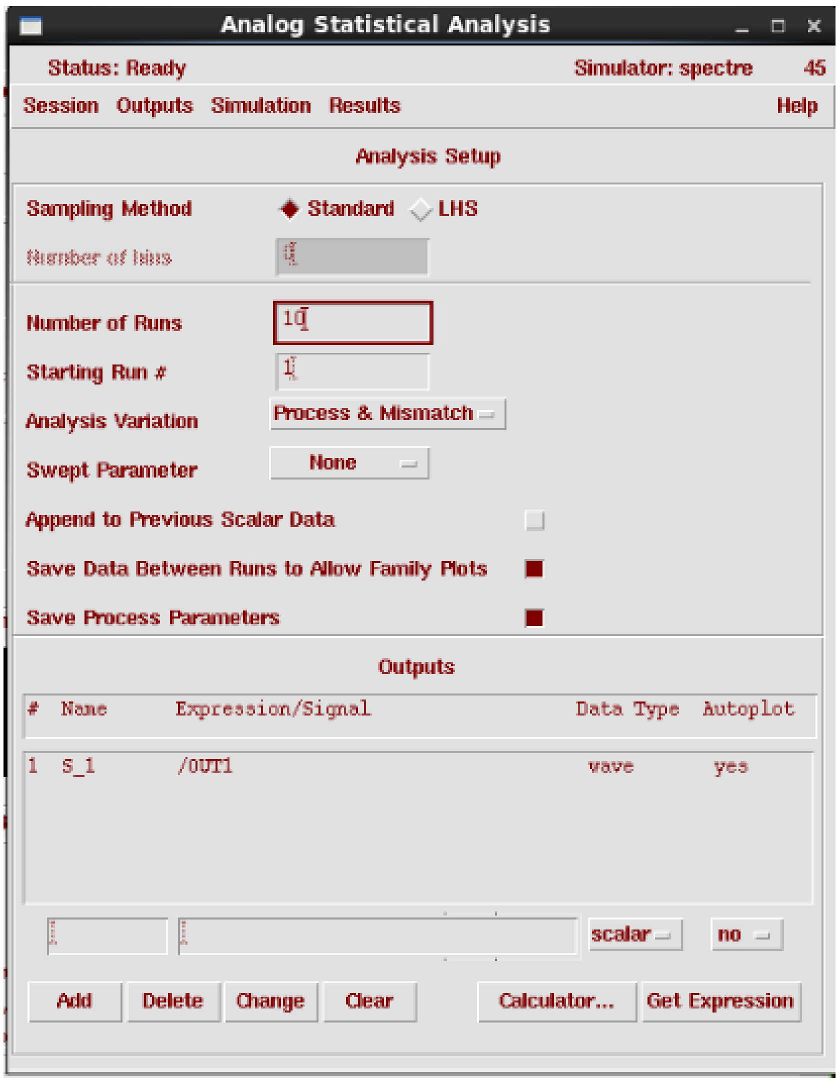
Figure A.16: Set simulation variables



Name (opt.):
Signal: /OUT1
Type: voltage
Will Be: [✓ Plotted] Saved Marched

#	Name/Signal/Expr	Value	Plot	Save	March
1	OUT1		yes	allv	no

Figure A.17: Configure plotted signals


Analog Statistical Analysis
Session Outputs Simulation Results Help
Add Delete Change Clear Calculator... Get Expression

Status: Ready
Simulator: spectre 45

Analysis Setup

Sampling Method [✓ Standard] LHS
Number of bins 0
Number of Runs 10
Starting Run # 1
Analysis Variation Process & Mismatch
Swept Parameter None
Append to Previous Scalar Data ☐
Save Data Between Runs to Allow Family Plots ■
Save Process Parameters ■

Outputs

#	Name	Expression/Signal	Data Type	Autoplot
1	S_1	/OUT1	wave	yes

scalar no

Figure A.18: Run Monte-Carlo Simulation

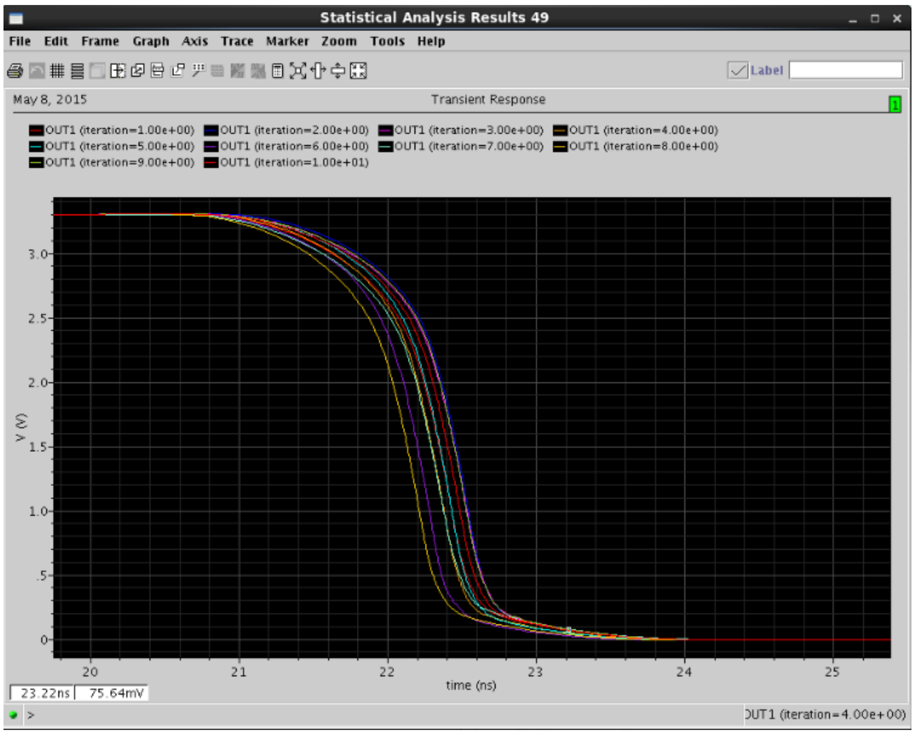


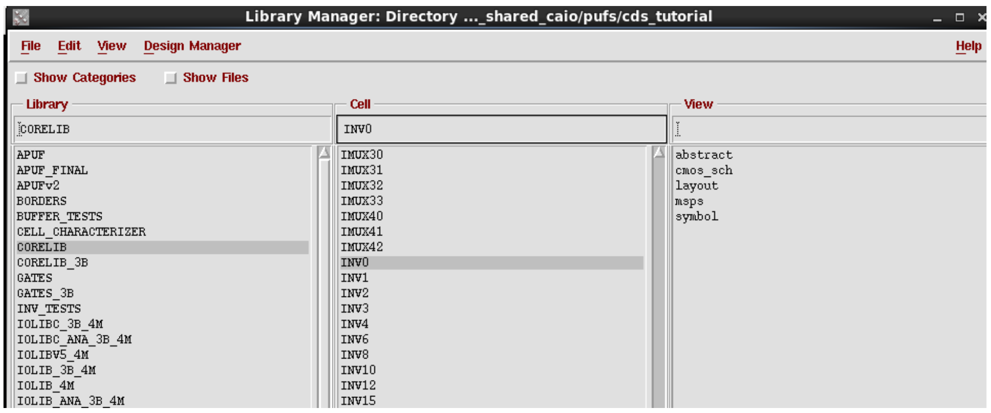
Figure A.19: Monte-Carlo result

A.3 Create and modify 'L' and 'W' of an inverter cell

In Section A.2 it was described the complete flow to perform a Monte-Carlo simulation using an existent "*inverter*" gate. In this section, it is explained how to copy an existent cell from the library provided by the foundry, perform geometric modifications in L and W, and verify the impact in terms of delay on the generated result.

First, it is necessary to copy the cell to be modified from the "*CORELIB*" library provided by the foundry to "*MCSIM_INV_GATE*". For that, selects the cell $CORELIB \rightarrow INV0$, as showed in Figure A.20, left-click on it and than selects "*copy*". In the opened window, configure the destination library as showed in Figure A.21 and press "OK". Than, a "*INV0*" cell should be available in the "*MCSIM_INV_GATE*" library. On this library, as showed in "Figure A.22", double-click the "*cmos_sch*" view to open the window. Than, open the cell properties, which should show the configured as showed in Figure A.23. Change the parameters related to L and W, keeping the same original values and adding the multipliers K_L and K_W, as showed in Figure A.24. Click *OK* and now the new gate is ready to be instantiated in the original project described in Section A.2. Open again the window showed in Figure A.8 and modify the instance properties to use the modified $INV0$ gate, as showed in Figure A.25. Perform this task for the three gates and than click in *check and save* to verify if there exists any errors during this task.

Finally, open the simulation window showed in Figure A.26, configure the new variables K_L and K_W, and perform the simulation again. Note that now, one is able to modify the geometric values of the cells from the simulation window.


Library Manager: Directory ..._shared_caio/pufs/cds_tutorial
File Edit View Design Manager Help
Show Categories Show Files

Library: CORELIB

```
APUF
APUF_FINAL
APUFv2
BORDERS
BUFFER_TESTS
CELL_CHARACTERIZER
CORELIB
CORELIB_3B
GATES
GATES_3B
INV_TESTS
IOLIBC_3B_4M
IOLIBC_ANA_3B_4M
IOLIBV5_4M
IOLIB_3B_4M
IOLIB_4M
IOLIB_ANA_3B_4M
```

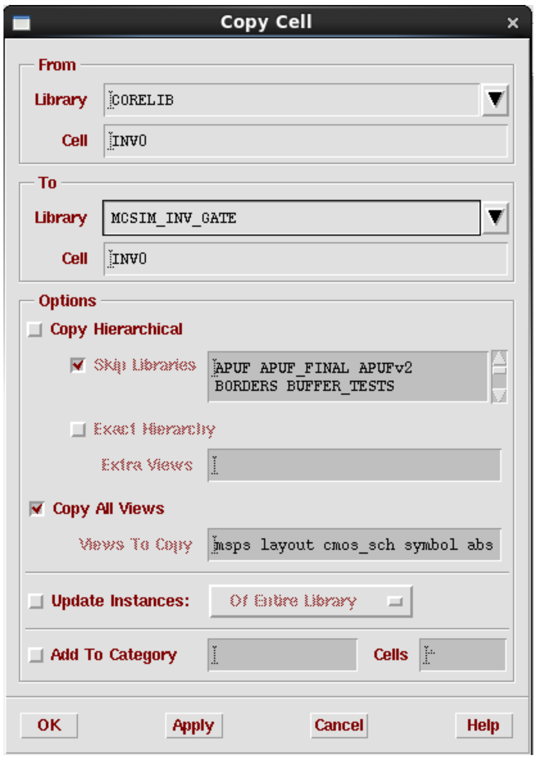
Cell: INV0

```
IMUX30
IMUX31
IMUX32
IMUX33
IMUX40
IMUX41
IMUX42
INV0
INV1
INV2
INV3
INV4
INV6
INV8
INV10
INV12
INV15
```

View:

```
abstract
cmos_sch
layout
msps
symbol
```

Figure A.20: Copy inv cell

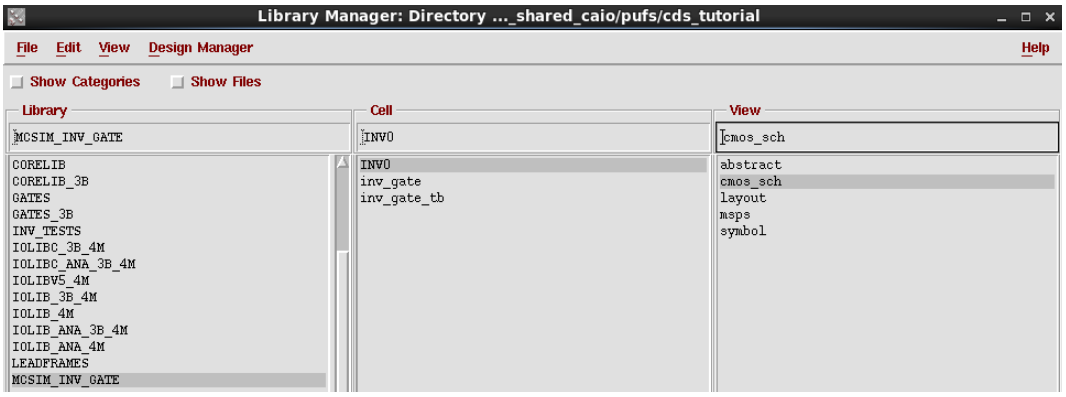


From
Library: CORELIB
Cell: INV0

To
Library: MCSIM_INV_GATE
Cell: INV0

Options
Copy Hierarchical [ ]
Skip Libraries [✓]: APUF APUF_FINAL APUFv2 BORDERS BUFFER_TESTS
Exact Hierarchy [ ]
Extra Views:
Copy All Views [✓]
Views To Copy: msps layout cmos_sch symbol abs
Update Instances: [ ] Of Entire Library
Add To Category: [ ] Cells:

Figure A.21: Copy inv cell window


Library Manager: Directory ..._shared_caio/pufs/cds_tutorial
File Edit View Design Manager Help
Show Categories Show Files

Library: MCSIM_INV_GATE

```
CORELIB
CORELIB_3B
GATES
GATES_3B
INV_TESTS
IOLIBC_3B_4M
IOLIBC_ANA_3B_4M
IOLIBV5_4M
IOLIB_3B_4M
IOLIB_4M
IOLIB_ANA_3B_4M
IOLIB_ANA_4M
LEADFRAMES
MCSIM_INV_GATE
```

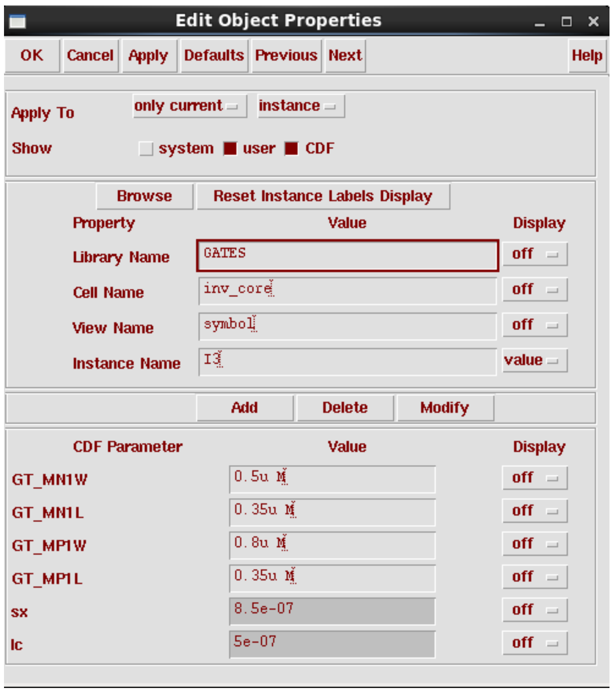
Cell: INV0

```
INV0
inv_gate
inv_gate_tb
```

View: cmos_sch

```
abstract
cmos_sch
layout
msps
symbol
```

Figure A.22: Inv cell added to library

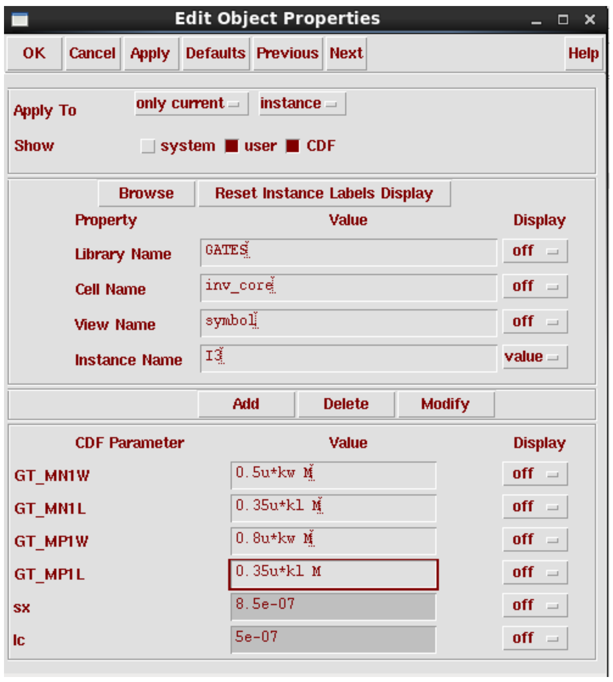


Apply To	only current	instance
Show	☐ system ■ user ■ CDF	

Property	Value	Display
Library Name	GATES	off
Cell Name	inv_core	off
View Name	symbol	off
Instance Name	I3	value

CDF Parameter	Value	Display
GT_MN1W	0.5u M	off
GT_MN1L	0.35u M	off
GT_MP1W	0.8u M	off
GT_MP1L	0.35u M	off
sx	8.5e-07	off
lc	5e-07	off

Figure A.23: Original inv cell properties

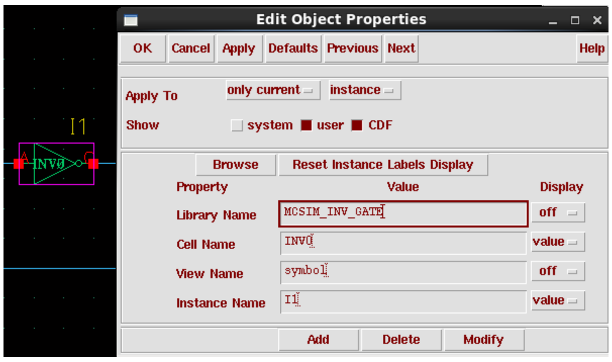


Apply To	only current	instance
Show	☐ system ■ user ■ CDF	

Property	Value	Display
Library Name	GATES	off
Cell Name	inv_core	off
View Name	symbol	off
Instance Name	I3	value

CDF Parameter	Value	Display
GT_MN1W	0.5u*kw M	off
GT_MN1L	0.35u*kl M	off
GT_MP1W	0.8u*kw M	off
GT_MP1L	0.35u*kl M	off
sx	8.5e-07	off
lc	5e-07	off

Figure A.24: Inv cell properties with K_L and K_W

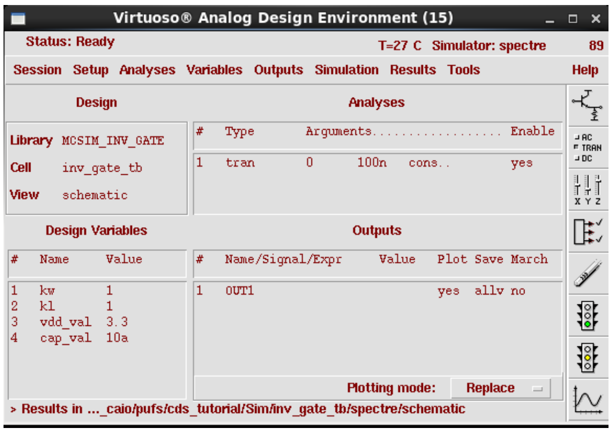


Apply To: only current, instance

Show: system, user, CDF

Property	Value	Display
Library Name	MCSIM_INV_GATE	off
Cell Name	INV0	value
View Name	symbol	off
Instance Name	I1	value

Figure A.25: Use the modified version of inv cell



Status: Ready T=27 C Simulator: spectre 89

Design

Library MCSIM_INV_GATE
Cell inv_gate_tb
View schematic

Analyses

#	Type	Arguments.................			Enable
1	tran	0	100n	cons..	yes

Design Variables

#	Name	Value
1	kw	1
2	kl	1
3	vdd_val	3.3
4	cap_val	10a

Outputs

#	Name/Signal/Expr	Value	Plot	Save	March
1	OUT1		yes	allv	no

Plotting mode: Replace

> Results in ..._caio/pufs/cds_tutorial/Sim/inv_gate_tb/spectre/schematic

Figure A.26: Define K_L, K_W and run the simulation

A.4 Ocean script

In Section A.2 and Section A.3 it was described the complete flow to perform a Monte-Carlo simulation using the graphical interface. After the circuit is simulated for the first time, the next simulations can be performed using the *SKILL* script language supported by Cadence, that runs in *Ocean* (*Open Command Environment for Analysis*). The Script A.2 shows how to run the Monte-Carlo through the usage of scripts. There exists many advantages in the usage of scripts for the simulation, in this case an extra level of control, not found in the graphical interface, can be performed. It comes to the definition of the "*seed*" parameter, described below:

> "This is the optional starting seed for the random number generator. By always specifying the same seed, you can reproduce a previous experiment. If you do not specify a seed, then each time that you run the analysis, you get different results; that is, a different stream of pseudorandom numbers is generated. If you do not specify a seed, the Spectre simulator uses the Spectre process id (PID) as a seed."

So, as it is required to always generate the same instances of chips during the simulations, this parameter is very important for the validity of the analysis. The other parameters are described as comments in the script.

Listing A.2: Ocean script to run inv simulation

```
ocnWaveformTool( 'wavescan )
simulator( 'spectre )

;model libraries
modelFile(
    '("/tools/techlib/hk380/spectre/c35/cmos53.scs"
                                            "cmosmc")
    '("/tools/techlib/hk380/spectre/c35/res.scs" "resmc")
    '("/tools/techlib/hk380/spectre/c35/cap.scs" "capmc")
    '("/tools/techlib/hk380/spectre/c35/bip.scs" "bipmc")
    '("/tools/techlib/hk380/spectre/c35/ind.scs" "indmc")
)

;geometry design multipliers
kl = 16
kw = 1

;load inv_gate_tb project
design("../cds_tutorial/Sim/inv_gate_tb/spectre/
                        schematic/netlist/netlist")
resultsDir( "../cds_tutorial/Sim/inv_gate_tb/
                                spectre/schematic" )
analysis('tran ?stop "200n")
desVar(   "vdd_val" 3.3)
desVar(   "cap_val" 0.01f)
desVar(   "kl" kl)
desVar(   "kw" kw)
envOption(
'paramRangeCheckFile "/tools/techlib/hk380/
                spectre/ams_range.lmts")
temp( 27 )

;Set MonteCarlo Configurations
monteCarlo( ?numIters "200" ?startIter "1" ?seed "1234"
    ?analysisVariation 'processAndMismatch
                        ?sweptParam "None"
    ?sweptParamVals "27" ?saveData t
    ?nomRun "yes" ?append nil
    ?saveProcessParams t
)

;Run MonteCarlo Simulation
monteRun()
```

```
;select result
selectResults('tran)

;Plot waveform object
newWindow()
addTitle( "INV0_CELL_CHARACTERIZER_kW=1")
;plot(v("IN"))
plot(v("OUT1"))
plot(v("OUT2"))

;Insert horizontal markers
awvPlaceYMarker(currentWindow() 1.65)

; exit
```

As it was necessary to perform several simulations during the full circuit simulation, the OCEAN script also supports to run several simulations at the same machine, using all cores, or even using a cluster. The Script A.3 contain the necessary modification to perform the "*distributed*" simulation mode.

Listing A.3: Ocean script to run distributed inv simulation

```
ocnWaveformTool( 'wavescan )
simulator( 'spectre )
hostMode( "distributed" )
design(  "../cds_tutorial/Sim/inv_gate_tb/spectre/schematic/
                                        netlist/netlist")
resultsDir( "../cds_tutorial/Sim/inv_gate_tb/spectre/
                                        schematic" )
modelFile(
    '("/tools/techlib/hk380/spectre/c35/cmos53.scs"
                                            "cmosmc")
    '("/tools/techlib/hk380/spectre/c35/res.scs" "resmc")
    '("/tools/techlib/hk380/spectre/c35/cap.scs" "capmc")
    '("/tools/techlib/hk380/spectre/c35/bip.scs" "bipmc")
    '("/tools/techlib/hk380/spectre/c35/ind.scs" "indmc")
)
; initialize jobList to nil
jobList = nil

;design constants
kl = 16
```

```
kw = 1

analysis('tran ?stop "200u"  )
desVar(   "cap_val" 0.01f       )
desVar(      "kl" kl     )
desVar(      "kw" kw     )
desVar(      "vdd_val" 3.3     )
envOption(
'paramRangeCheckFile  "/tools/techlib/hk380/
                          spectre/ams_range.lmts"
)
temp( 27 )

;Set MonteCarlo Configurations
monteCarlo( ?numIters "200" ?startIter "1"  ?seed "1234"
    ?analysisVariation 'processAndMismatch
    ?sweptParam "None"
    ?sweptParamVals "27" ?saveData t
    ?nomRun "yes" ?append nil
    ?saveProcessParams t
)
;Run MonteCarlo Simulation
;monteRun()
job1 = monteRun(?queue "fast")
; wait for the job to finish
wait( job1 )
openResults( job1 )
;select result
selectResults('tran)

;get corresponding values in text mode
ocnPrint(?output "../../cds_tutorial/final_results/
        inv_gate/kl16/result_inv_gate_000" v("OUT2"))

;Analog to digital conversion
;OUT2
out_dig_out2=awvAnalog2Digital(VT("OUT2") 1.8 0.2 0 0
                                            "hilo")
ocnPrint(out_dig_out2)
ocnPrint(?output "../../cds_tutorial/final_results/
        inv_gate/kl16/result_dig_inv_gate_000" out_dig_out2)

exit
```

Appendix B

Tutorial: Monte-Carlo Simulation Framework

In this chapter it is explained the framework created to manage all the necessary simulations, whose results are described in the previous sessions. The objective is to provide the required knowledge for the ones interested in reproduce the environment using the same tools as used in this work.

The tree below shows the directory structure that contains the necessary scripts for the process. The ocean scripts, available in folder "*ocean_scripts*", are generated proportional to each line in file "*stimulus0.in*". In this example, there exists two challenges in "*stimulus0.in*", generating output files "*apuf64_dfc1_xor20_tb_kL32_000_000.ocn*" and "*apuf64_dfc1_xor20_tb_kL32_001_000.ocn*".

The content of "*stimulus0.in*" in this example is shown below.

```
1110001101000101110001010001101001011101010001010100001010011100
1110001101000101110001010001101001011101010001010100001010011101
```

```
|   input_challenge
|   |__gen_challenge.py
|   |__stimulus0.in
|__ocean_scripts
|   |__apuf64_dfc1_xor20_tb_kL32_000_000.ocn
|   |__apuf64_dfc1_xor20_tb_kL32_001_000.ocn
|__src
    |__gen_ocean_script
    |   |__gen_ocean_main.py
    |__main.py
    |__run_ocean
        |__run_ocean_main.py
```

The Script B.1 is the main script of the framework. In it, one will configure the (a) number of instances ("*numIters*"), (b) number of the first instance to be simulated ("*startIter*"), (c) source of entropy ("*seed*") , (d) simulation temperature ("*temp*"), (e) path of the netlist ("*sim_pathname*"), (f) path where result will be stored ("*result_pathname*"), (g)

K_L multiplier ("*kl*"), (h) K_W multiplier ("*kw*") and (i) number of simulation to be run in parallel, usually the number of *cores* in the computer's processor ("*n_paralel_sim*").

Listing B.1: main.py

```
'''
Created on Oct 27, 2013

@author: jrodrigo
'''
# import <other_scritps>
import gen_ocean_script.gen_ocean_main
import run_ocean.run_ocean_main

# Run all procedures to perform MonteCarlo Simulation in
# Analog Environment

#Configurations
config = {
  'numIters' : '32',
  'startIter' : '1',
  'seed' : '1234',
  'temp' : '27',
  'sim_pathname' : 'apuf64_dfc1_xor20_tb',  #name of working
                  #directory where the netlist was generated
  'result_pathname' : 'apuf_final_results/
                        apuf64_dfc1_xor20_results/kl32', #name
                        #of directory where results will be
                        #stored "ex: mc_results/apuf64"
  'kl' : '32', #multiplier of transistors in L
  'kw' : '1', #multiplier of transistors in W
  'n_paralel_sim' : '32'
}


#1- Generate ocean scripts based on challenges given in
#"input_challenge"
path = '../'
gen_ocean_script.gen_ocean_main.gen_ocean(path, config)

#2- Perform MonteCarlo simulation for every ocean script
# created.
path = '../'
run_ocean.run_ocean_main.run_ocean(path, config)
```

The Script B.2 implements the process of converting the challenges from the digital format

(0's and 1's) to the representation in analog voltage. Taking as example an APUF circuit with 3 stages, and the challenge is "1011", the analog conversion and line to be in ocean script is described below. Plus, it also configure the remaining of every ocean script with the parameters defined in "*main.py*".

```
desVar("sel_0"  3.3)
desVar("sel_1"  0  )
desVar("sel_2"  3.3)
desVar("sel_3"  3.3)
```

Listing B.2: gen_ocean_main.py

```
'''
Created on Oct 27, 2013

@author: jrodrigo
'''
#!/usr/bin/python
#
# Author: Jefferson Rodrigo Capovilla (jefcap@gmail.com)
#
# Description: Generate ocean scripts according to number of
#              challenges passed as input
#
# Date: 2013-10-27
#
# License: Attribution-NonCommercial-ShareAlike 3.0 Unported
#          (CC BY-NC-SA 3.0)
#===========================================================

import os,sys
import string
import copy
from glob import glob


def write_header (file_handler, config):
 header_info = []
 string = """
ocnWaveformTool( 'wavescan )
simulator( 'spectre )
hostMode( "distributed" )
design(     "../../cds_tutorial/Sim/%(sim_pathname)s/spectre/
             schematic/netlist/netlist")
resultsDir( "../../cds_tutorial/Sim/%(sim_pathname)s/spectre/
            schematic" )
```

```
modelFile(
    '("/tools/techlib/hk380/spectre/c35/mcparams.scs" "")
    '("/tools/techlib/hk380/spectre/c35/cmos53.scs" "cmosmc")
    '("/tools/techlib/hk380/spectre/c35/res.scs" "resmc")
    '("/tools/techlib/hk380/spectre/c35/cap.scs" "capmc")
    '("/tools/techlib/hk380/spectre/c35/bip.scs" "bipmc")
    '("/tools/techlib/hk380/spectre/c35/ind.scs" "indmc")
)
; initialize jobList to nil
jobList = nil

;design constants
kl = %(kl)s
kw = %(kw)s

analysis('tran ?stop "500u"  )
desVar(   "cap_val" 1f  )
desVar(      "kl" kl     )
desVar(      "kw" kw     )
desVar(      "vdd_val" 3.3     )
"""

 string_filled=string%{'sim_pathname':config['sim_pathname'],
                'kl': config['kl'], 'kw': config['kw'],}
 #header_info.append(string)
 #header_info_str =  ''.join(header_info)
 file_handler.write(string_filled)


def write_challenge(file_handler, stim):
 challenge_info = []
 for i in range(0,len(stim)-1):
  bit = stim[i]
  if bit == '1':
   vcc_val = ' 3.3 '
  else:
   vcc_val = ' 0 '
  string = 'desVar(      "sel_'+ str(i) +'" '+vcc_val+'  )\n'
  challenge_info.append(string)

 challenge_info_str =  ''.join(challenge_info)
 file_handler.write(challenge_info_str)
 pass
```

```
def write_tail(file_handler, file_number, config,
                                subgroup_number):
 tail_info = []
 string = """
;saveOption( ?outputParamInfo t )
;saveOption( ?elementInfo t )
;saveOption( ?modelParamInfo t )
;saveOption( ?saveahdlvars "all" )
;saveOption( 'useprobes "yes" )
;saveOption( 'currents "all" )
;saveOption( 'pwr "all" )
;saveOption( 'save "all" )
envOption(
 'paramRangeCheckFile  "/tools/techlib/hk380/spectre/
                                      ams_range.lmts"
)
temp( %(temp)s )

;Set MonteCarlo Configurations
monteCarlo( ?numIters "%(numIters)s"
    ?startIter "%(startIter)s"  ?seed "%(seed)s"
    ?analysisVariation 'processAndMismatch ?sweptParam "None"
    ?sweptParamVals "%(temp)s" ?saveData t
    ?nomRun "yes" ?append nil
    ?saveProcessParams t
)

;Run MonteCarlo Simulation
;monteRun()
job1 = monteRun(?queue "fast")

; wait for the job to finish
wait( job1 )
openResults( job1 )

;select result
selectResults('tran)

;Plot waveform object
;plot(v("IN"))
;plot(v("D"))
;plot(v("C"))
;plot(v("q_latch"))
```

```
;get corresponding values in text mode
;ocnPrint(?output "../../cds_tutorial/%(result_pathname)s/
result_in%(file_number)s_%(subgroup_number)s" v("IN"))
;ocnPrint(?output "../../cds_tutorial/%(result_pathname)s/
result_d%(file_number)s_%(subgroup_number)s" v("D"))
;ocnPrint(?output "../../cds_tutorial/%(result_pathname)s/
result_c%(file_number)s_%(subgroup_number)s" v("C"))
;ocnPrint(?output "../../cds_tutorial/%(result_pathname)s/
result_out_ff%(file_number)s_%(subgroup_number)s"
                                  v("out_ff"))
;ocnPrint(?output "../../cds_tutorial/%(result_pathname)s/
result_out_latch%(file_number)s_%(subgroup_number)s"
                                      v("q_latch"))

;Analog to digital conversion
;IN
out_dig_in=awvAnalog2Digital(VT("IN") 1.8 0.2 0 0 "hilo")
ocnPrint(out_dig_in)
ocnPrint(?output "../../cds_tutorial/%(result_pathname)s/
result_dig_in%(file_number)s_%(subgroup_number)s" out_dig_in)
;D
out_dig_d=awvAnalog2Digital(VT("D") 1.8 0.2 0 0 "hilo")
ocnPrint(out_dig_d)
ocnPrint(?output "../../cds_tutorial/%(result_pathname)s/
result_dig_d%(file_number)s_%(subgroup_number)s" out_dig_d)
;C
out_dig_c=awvAnalog2Digital(VT("C") 1.8 0.2 0 0 "hilo")
ocnPrint(out_dig_c)
ocnPrint(?output "../../cds_tutorial/%(result_pathname)s/
result_dig_c%(file_number)s_%(subgroup_number)s" out_dig_c)
;OUT_FF
;out_dig_out_ff=awvAnalog2Digital(VT("out_ff")
                            1.8 0.2 0 0 "hilo")
;ocnPrint(out_dig_out_ff)
;ocnPrint(?output "../../cds_tutorial/%(result_pathname)s/
result_dig_out_ff%(file_number)s_%(subgroup_number)s"
                              out_dig_out_ff)
;Q_LATCH
out_dig_out_latch=awvAnalog2Digital(VT("q_latch")
                          1.8 0.2 0 0 "hilo")
ocnPrint(out_dig_out_latch)
ocnPrint(?output "../../cds_tutorial/%(result_pathname)s/
result_dig_out_latch%(file_number)s_%(subgroup_number)s"
```

```
                                   out_dig_out_latch)

exit
"""

 string_filled = string%{'temp':config['temp'],
         'numIters':config['numIters'],
         'result_pathname':config['result_pathname'],
         'startIter':config['startIter'],
         'seed':config['seed'],
         'file_number':str(file_number).rjust(3,'0'),
         'subgroup_number': str(subgroup_number)
                                 .rjust(3,'0')}

 file_handler.write(string_filled)



def gen_ocean(path, config):

 print ("Generating Ocean Scripts...")

 filename = path + "input_challenge/stimulus0.in"
 try:
  stimulus = open(filename, "r")
 except AssertionError:
  print("Stimulus0.in file not found in
              ../../input_challenge/")
  sys.exit("ValueError")


 #read stimulus vectors from input file
 stimulus_list = []
 for line in stimulus:
   stimulus_list.append(line)
 stimulus.close()

 #evaluate stimulus_size (same as APUF size)
 apuf_size = len(stimulus_list[0]) -1

 #generate ocean script files
 file_number = 0  #challenge number
 original_config = copy.deepcopy(config)
 for stim in stimulus_list:
```

```
#generating subgroups of scripts according to
#'n_paralel_sim'
subgroup_config = {} # dictionary to store config of
                      # each subgroup

if (int(original_config['numIters']) %
        int(original_config['n_paralel_sim']) == 0):
 #number of subgroups files to be generate do
 #accord n_parallel_sim
 n_of_subgroups = int(original_config['numIters']) /
      int(original_config['n_paralel_sim'])


 for i in range (0, int(n_of_subgroups)):
  config['numIters'] = original_config['n_paralel_sim']
  config['startIter'] = i *
             int(original_config['n_paralel_sim']) + 1
  subgroup_config[i] = copy.deepcopy(config)
else:
 n_of_subgroups = (int(original_config['numIters']) /
      int(original_config['n_paralel_sim'])) + 1
 for i in range (0, int(n_of_subgroups)-1):
  config['numIters'] = original_config['n_paralel_sim']
  config['startIter'] = i *
           int(original_config['n_paralel_sim']) + 1
  subgroup_config[i] = copy.deepcopy(config)
 config['numIters'] = int(original_config['numIters'])
                    % int(original_config['n_paralel_sim'])
 config['startIter'] = (i+1) *
                 int(original_config['n_paralel_sim']) + 1
 subgroup_config[i+1] = copy.deepcopy(config)


subgroup_number = 0
for config_number in subgroup_config:
 #filename = path + "ocean_scripts/MonteCarlo_apuf"
 #+str(apuf_size)+"_stages"+str(file_number).rjust(3,'0')+
 #"_"+str(subgroup_number).rjust(3,'0')+".ocn"
 filename = path + "ocean_scripts/"+config['sim_pathname']+
 "_"+"kL"+config['kl']+"_"+str(file_number).rjust(3,'0')+
 "_"+str(subgroup_number).rjust(3,'0')+".ocn"

 ocean_file = open(filename, "w")
```

```
      #Write header
      write_header(ocean_file, subgroup_config[config_number])

      #write challenge values
      write_challenge(ocean_file, stim)

      #write tail
      write_tail(ocean_file, file_number,
         subgroup_config[config_number], subgroup_number)

      #close ocean file
      ocean_file.close()

      subgroup_number = subgroup_number +1

     file_number = file_number + 1


   print ("Done!")



if __name__ == "__main__":

    #Set filepath
    path = '../../'

    #Configurations
    config = {
      'numIters' : '257',
      'startIter' : '1',
      'seed' : '1234',
      'temp' : '27',
      'n_paralel_sim' : '32'
    }

    #call function
    gen_ocean(path, config)
```

In Script B.3 it is listed all the ocean files generated by *gen_ ocean_ main.py*, and an *bash script* is created. For the same example, supposing only two challenges, the bash script generated is showed below. The advantage of using a bash script for running the simulations is that it is very easy to continue the process if, for some reason, the simulation has stopped. To continue, one just needs to check the last result stored in the output folder, and restart the simulation from that point, by commenting the previous lines in the bash script.

```
#!/bin/bash
time ocean -restore apuf64_dfc1_xor20_tb_kL32_000_000.ocn;
rm -rf ../../cds_tutorial/Sim/apuf64_dfc1_xor20_tb/spectre/
       schematic/monteCarlo/job*
time ocean -restore apuf64_dfc1_xor20_tb_kL32_001_000.ocn;
rm -rf ../../cds_tutorial/Sim/apuf64_dfc1_xor20_tb/spectre/
       schematic/monteCarlo/job*
```

Listing B.3: run_ocean_main.py

```
'''
Created on Oct 28, 2013

@author: jrodrigo
'''
#!/usr/bin/python
#
# Author: Jefferson Rodrigo Capovilla (jefcap@gmail.com)
#
# Description: Run ocean scripts on folder 'ocean_scripts'
#
# Date: 2013-10-27
#
# License: Attribution-NonCommercial-ShareAlike 3.0 Unported
#          (CC BY-NC-SA 3.0)
#============================================================

import os,sys
from subprocess import *
from glob import glob
import shutil


def run_ocean(path, config):
 print ("Running Ocean Scripts...")

 #listing all ocean scripts:
 path = os.path.dirname(__file__)
 print (path)
 files = sys.argv[1:]
 if not files:
  files = glob(os.path.join(path,'../../ocean_scripts',
                                    '*.ocn'))
  files.sort()
 #extract .ocn filenames
 filename_list = []
 for ocean_file in files:
  filename_list.append(os.path.split(ocean_file)[-1])

 #generate bash_script to run all ocean scripts
 print("----------------------------")
 print("#!/bin/bash")
 for filename in filename_list:
  print ("time ocean -restore " + filename +"; ")
  string = "rm -rf ../../cds_tutorial/Sim/%(sim_pathname)s/
            spectre/schematic/monteCarlo/job*"
  string_filled = string%{'sim_pathname':
                  config['sim_pathname']}

  print (string_filled)

 print ("echo Done!")



if __name__ == "__main__":

 #Set filepath
 path = '../../'

 #call function
 run_ocean(path)
```