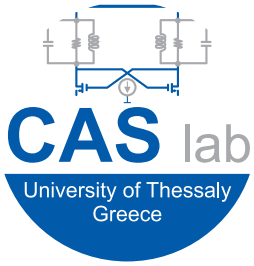


DIPLOMA THESIS

VLSI Circuit Component Variation Investigation & Optimisation Using Monte-Carlo Methodologies

Student:
Evangelos Bakas
evabakas@uth.gr

Supervisor:
Christos Sotiriou
chsotiriou@e-ce.uth.gr
Committee:
Georgios Stamoulis
georges@e-ce.uth.gr
Fotios Plessas
fplessas@e-ce.uth.gr

A thesis submitted in fulfillment of the requirements
for the degree of Integrated Master
in the
Circuits & Systems Laboratory (CASlab)
Department of Electrical and Computer Engineering

July 11, 2023

Acknowledgements

First and foremost, I would like to thank Prof. Christos Sotiriou for his help and guidance in this thesis and also for the precious experience I obtained during my time in CASlab. I would also like to thank Prof. Georgios Stamoulis and Prof. Fotios Plessas for their presence in my thesis presentation, as well as their contributions to my academic knowledge through their courses.

Furthermore, I would like to express my sincere appreciation and gratitude to the other CASlab members and namely PhD candidates Nikolaos Blias and Stavros Simoglou and undergraduate student Iordanis Lilitsis for the time they spent and the invaluable support and knowledge they provided me for this thesis.

Finally, my humble thanks to all my friends and family who supported me all these years throughout my undergraduate academic years.

Evangelos Bakas
Volos, 2023

DISCLAIMER ON ACADEMIC ETHICS AND INTELLECTUAL PROPERTY RIGHTS

«Being fully aware of the implications of copyright laws, I expressly state that this diploma thesis, as well as the electronic files and source codes developed or modified in the course of this thesis, are solely the product of my personal work and do not infringe any rights of intellectual property, personality and personal data of third parties, do not contain work / contributions of third parties for which the permission of the authors / beneficiaries is required and are not a product of partial or complete plagiarism, while the sources used are limited to the bibliographic references only and meet the rules of scientific citing. The points where I have used ideas, text, files and / or sources of other authors are clearly mentioned in the text with the appropriate citation and the relevant complete reference is included in the bibliographic references section. I also declare that the results of the work have not been used to obtain another degree. I fully, individually and personally undertake all legal and administrative consequences that may arise in the event that it is proven, in the course of time, that this thesis or part of it does not belong to me because it is a product of plagiarism».

The declarant

Evangelos Bakas

ΥΠΕΥΘΥΝΗ ΔΗΛΩΣΗ ΠΕΡΙ ΑΚΑΔΗΜΑΪΚΗΣ ΔΕΟΝΤΟΛΟΓΙΑΣ ΚΑΙ ΠΝΕΥΜΑΤΙΚΩΝ ΔΙΚΑΙΩΜΑΤΩΝ

«Με πλήρη επίγνωση των συνεπειών του νόμου περί πνευματικών δικαιωμάτων, δηλώνω ρητά ότι η παρούσα διπλωματική εργασία, καθώς και τα ηλεκτρονικά αρχεία και πηγαίοι κώδικες που αναπτύχθηκαν ή τροποποιήθηκαν στα πλαίσια αυτής της εργασίας, αποτελεί αποκλειστικά προϊόν προσωπικής μου εργασίας, δεν προσβάλλει κάθε μορφής δικαιώματα διανοητικής ιδιοκτησίας, προσωπικότητας και προσωπικών δεδομένων τρίτων, δεν περιέχει έργα/εισφορές τρίτων για τα οποία απαιτείται άδεια των δημιουργών/δικαιούχων και δεν είναι προϊόν μερικής ή ολικής αντιγραφής, οι πηγές δε που χρησιμοποιήθηκαν περιορίζονται στις βιβλιογραφικές αναφορές και μόνον και πληρούν τους κανόνες της επιστημονικής παράθεσης. Τα σημεία όπου έχω χρησιμοποιήσει ιδέες, κείμενο, αρχεία ή/και πηγές άλλων συγγραφέων, αναφέρονται ευδιάκριτα στο κείμενο με την κατάλληλη παραπομπή και η σχετική αναφορά περιλαμβάνεται στο τμήμα των βιβλιογραφικών αναφορών με πλήρη περιγραφή. Δηλώνω επίσης ότι τα αποτελέσματα της εργασίας δεν έχουν χρησιμοποιηθεί για την απόκτηση άλλου πτυχίου. Αναλαμβάνω πλήρως, ατομικά και προσωπικά, όλες τις νομικές και διοικητικές συνέπειες που δύναται να προκύψουν στην περίπτωση κατά την οποία αποδειχθεί, διαχρονικά, ότι η εργασία αυτή ή τμήμα της δεν μου ανήκει διότι είναι προϊόν λογοκλοπής».

Ο/Η Δηλών/ούσα

Ευάγγελος Μπάκας

Abstract

Process variation in digital circuits has been an important issue in the semiconductor industry over the years. These naturally occurring phenomena may impact the timing and functionality of the design, leading to unexpected timing violations. To deal with these issues, the industry has tried many different approaches in the field of Stating Timing Analysis (STA). While the traditional corner-based timing analysis (i.e. typical, slow, fast) might solve the problem by calculating the worst-case scenario and adjusting the circuit delay to avoid setup and hold violations, it may also lead to unnecessary over-constraining of the design, since the probability of worst-case scenario occurrence is relatively low. In situations like these, the use of Monte-Carlo methodologies is preferred, to calculate the relative behaviour of the design out of many different random situations. As such, there is a possibility to prevent timing violations, while also avoiding burdening the design functionality due to over-constraining. In this thesis, we explore the effects of process variation in the timing of simple digital circuits by performing Monte-Carlo simulations at the transistor level using the SPICE-compatible Cadence Spectre tool. Finally, we present the results of our experiments to explain and analyse the effects of these variations in the aforementioned designs.

Περίληψη

Η διακύμανση στο χρονισμό των ψηφιακών κυκλωμάτων αποτελεί ένα σημαντικό ζήτημα στη βιομηχανία ημιαγωγών ανά τα χρόνια. Αυτά τα φαινόμενα μπορούν να επηρεάσουν όχι μόνο το χρονισμό αλλά και τη λειτουργικότητα του συστήματος προς σχεδιασμό, οδηγώντας σε απροσδόκητα σφάλματα και βλάβες. Για την αντιμετώπιση αυτών των ζητημάτων, η βιομηχανία έχει δοκιμάσει πολλές διαφορετικές προσεγγίσεις στον τομέα της Στατικής Χρονικής Ανάλυσης (STA) κυκλωμάτων. Ενώ η ευρέως διαδεδομένη corner-based ανάλυση χρονισμού (typical, slow, fast) ενδέχεται να επιλύσει το πρόβλημα υπολογίζοντας το χειρότερο δυνατό σενάριο και προσαρμόζοντας την καθυστέρηση του κυκλώματος, ώστε να αποφευχθούν οι setup και hold παραβιάσεις, μπορεί ταυτόχρονα να οδηγήσει και σε περιττό υπερ-περιορισμό του σχεδιασμού, καθώς η πιθανότητα εμφάνισης του χειρότερου σεναρίου είναι σχετικά χαμηλή. Σε τέτοιες περιπτώσεις, προτιμάται η χρήση Monte-Carlo μεθοδολογιών, προκειμένου να υπολογιστεί η σχετική συμπεριφορά του κυκλώματος από πολλές διαφορετικές τυχαίες καταστάσεις. Συνεπώς, υπάρχει η δυνατότητα αποτροπής των παραβιάσεων χρονισμού, ενώ παράλληλα αποφεύγεται η περαιτέρω επιβάρυνση της λειτουργικότητας του συστήματος προς σχεδιασμό λόγω υπερβολικών περιορισμών. Στην παρούσα εργασία, διερευνούμε τις επιπτώσεις των διακυμάνσεων στο χρονισμό απλών ψηφιακών κυκλωμάτων εφαρμόζοντας προσομοιώσεις Monte-Carlo σε επίπεδο τρανζίστορ, χρησιμοποιώντας το, συμβατό με τη γλώσσα του SPICE, εργαλείο Spectre της Cadence. Τέλος, παρουσιάζουμε τα αποτελέσματα των πειραμάτων μας για να εξηγήσουμε και να αναλύσουμε τις συνέπειες αυτών των διακυμάνσεων στα προαναφερθέντα κυκλώματα.

VLSI Circuit Component Variation Investigation & Optimisation Using Monte-Carlo Methodologies

Evangelos Bakas
evabakas@uth.gr

Έρευνα και Βελτιστοποίηση Διακύμανσης Στοιχείων VLSI Κυκλωμάτων με τη Μεθοδολογία Monte-Carlo.

Ευάγγελος Μπάκας
evabakas@uth.gr

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Variations in VLSI digital circuits is one of the most relevant matters in the modern semiconductor industry. The ability to calculate the behaviour of the chip before the fabrication concerns both foundries and academic research groups. Depending on the type of each variation, a different approach is applied and a different group of experts is advised. For example, the resistor tolerance variation is different than a timing variation in a digital circuit. As such, it is difficult to determine all the causes of random variations at once, since an extensive research of each factor is required.

In the following thesis, we will discuss the basic process variations in VLSI digital circuits and examine how their timing is affected in multiple random situations, using the Monte-Carlo computational algorithm. This chapter contains the introduction and the main goal of this work. In chapter 2 we will describe the background knowledge needed in order to understand our work and in chapter 3 we will mention relevant works in the literature or industry, as well as the industrial tools used for this thesis. Chapters 4 and 5 describe our proposed work and the experimental results respectively. Finally, in chapter 6, we summarise our final conclusions and briefly describe possible future work on this topic.

1.1 Aim of this Work

The main purpose of this work is to investigate the effects of process variation in the timing of simple digital circuits. In the field of Static Timing Analysis (STA), the most common method to evaluate whether timing violations are present in the design is the corner-based analysis, in which the worst, best and typical cases are examined. However, the possibility of occurrence for each one of these cases is not calculated by this method and in most cases, the worst case scenario is rare. Due to this rarity, if we constrain the design according to the worst case in order to avoid timing violations, it will most likely end up over-constrained and will work much slower than intended. In order to avoid this scenario, the necessity to examine multiple iterations to have a more well-rounded view of the timing variations of the design has arisen.

In the industry, there are many different variations experts, which investigate

these phenomena in both lower and higher levels of chip design. For this work, we will only focus on the variations occurring in the transistor level digital circuits, using SPICE compatible tools. After the theoretical background information, we will present our own investigation, along with our experimental results and observations and conclude with our possible future expansions of this work.

1.2 Main Flow of this Work

To examine different outcomes of variation in transistor level circuits, we created a bash flow which generates automated SPICE decks of buffer and inverter chains, calls the circuit simulator to perform Monte-Carlo simulations and illustrates the delay and slew variations of the simulations in graphical histograms. The generation of the SPICE decks and the results presentation is made using **Python** scripts directly executed from the main bash script.

To apply timing variations in the designs under investigation, we generated a set of pseudo-random numbers from a specified probability distribution, to be used as derate factors of specific transistor parameters.

Chapter 2

Background

In this chapter, we will present the necessary background knowledge for our work. We will cover the basic terms of numerical analysis and the Monte-Carlo method, a simple introduction to the field of Static Timing Analysis (STA) in VLSI digital circuits and a fundamental review of the SPICE language and how process variations affects the semiconductor industry.

2.1 Mathematical Background

We will begin by explaining and presenting the essential mathematical background used for the purpose of this thesis. The main study covered is **numerical analysis** and, more specifically, the **Monte-Carlo method**, which is the main algorithm for our research work.

2.1.1 Numerical Analysis

Numerical analysis is a field of mathematics that relies on numerical approximation, instead of symbolic expressions, to solve continuous problems. These numerical methods are useful in many cases and problems, where the exact result is either impossible or extremely difficult to calculate using traditional equations [1]. Moreover, numerical analysis, not only gives approximate but accurate solutions, but also provides general characteristics of each method, such as result accuracy, convergence and computational complexity. Thus, numerical analysis is widely recognised and used in many different scientific fields, including engineering and economics. The most common tool associated with applying numerical analysis in various problems is MATLAB by MathWorks, along with its open-source counterpart Octave and many application specific tools, such as the one used in this work, which we will describe in 3.2.

To further understand the concept of approximation, we will briefly describe one of the most common category of numerical analysis methods, which are the **iterative** equation solving algorithms, e.g. the Newton's method. The main characteristic of these methods is that, in contrast to direct equation solving methods, they are not expected to terminate after a finite number of steps and the result of each iteration,

starting from an initial guess, further converges to the actual result, but without precisely reaching it [2].

To summarise, the field of numerical analysis is one of the most widely-used sections of mathematics, due to its large amount of applications in the majority of the sciences with many numerical methods having different uses in complex problem solving. The Monte-Carlo method is one of these algorithms, which is the one we used for the purpose of this work and we will explain in the next subsection.

2.1.2 Monte-Carlo Method

In the field of numerical analysis, a **Monte-Carlo method** is a computational algorithm that relies on repeated random samples to obtain numerical results [3]. It is mostly used for solving problems which contain randomness, e.g. the tolerance of a resistor, and as a result, they are difficult to produce a stable solution. In microelectronics engineering, Monte-Carlo methods are used to examine and analyse the variations present in analog and digital circuits. As such, many industrial tools in this field support Monte-Carlo simulations, including the tool used for this work.

While many different variations of Monte-Carlo methods exist, the general pattern follows these simple steps [3]:

1. From a specified domain of inputs, generate them randomly from a **probability distribution**.
2. Run the desired experiment using each of these random inputs, e.g. in this case the circuit simulations.
3. Aggregate the results.

A **probability distribution** is a mathematical function that presents the range of all the different values, along with their frequency of occurrence, a random variable can have. Within this range, the appearance likelihood of each value is determined by a number of factors, such as the **mean** value and the **standard deviation** [4]. Each probability distribution is defined by a unique **Probability Density Function (PDF)**, which graphically depicts the likelihood of an outcome for a specific random variable [5]. PDFs can be used for both discrete and continuous data, however, in this work, we will focus on the latter. For further understanding of these terms, we will describe three of the most common probability distributions along with their PDFs below.

Note: All the graphs were designed using *StatDist* online tool [6].

Uniform Distribution

The term **uniform distribution** refers to a probability distribution, in which all the possible outcomes of a given range have an equal chance to occur. The range of the distribution is defined by two parameters a and b, which define the lower and upper bounds of the range respectively, while the interval can be either open or

closed. The uniform distribution can be described by the following PDF:

$$f(x) = \begin{cases} \frac{1}{b-a} & a \leq x \leq b \\ 0 & x \leq a \text{ or } x \geq b \end{cases} \tag{2.1}$$

Assuming $a = 2$ and $b = 6$, figure 2.1 shows the graphical representation of the uniform distribution probability density function.

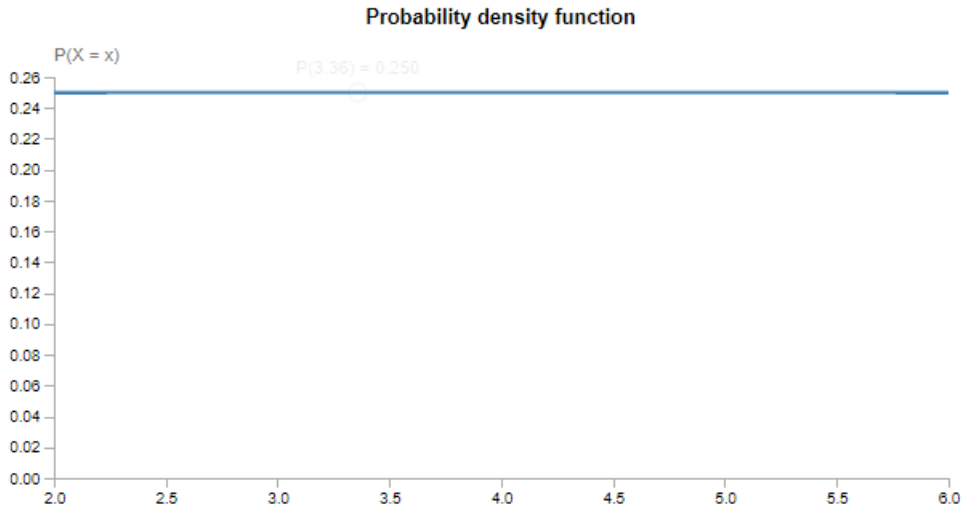


Figure 2.1: Uniform Distribution PDF with $a = 2$ and $b = 6$

By generating 5000 random samples, we can confirm the PDF, as shown in the following figure, where all the variables have an approximate equal occurrence frequency:

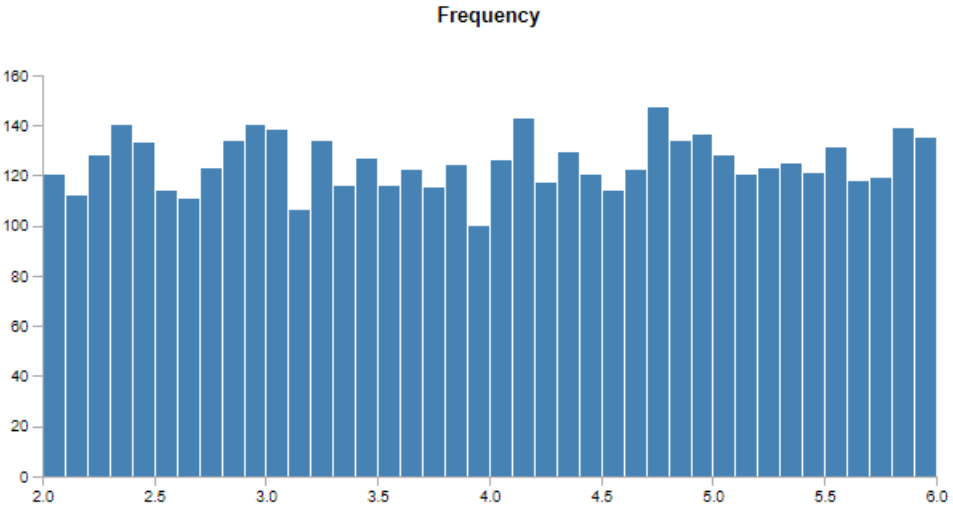


Figure 2.2: Uniform Distribution Samples with $a = 2$ and $b = 6$

Normal Distribution

A **normal** or **Gaussian distribution** is one of the most common type of distribution assumed in statistical analyses [7]. The main characteristic of the normal distribution is its **symmetry around the specified mean value**, which describes that, unlike the uniform distribution where all the data have an equal frequency of occurrence, the variables close to the mean value have a higher probability to occur than the ones further from the mean. In other words, the more we deviate from the mean, the less likely the respective data will occur.

In order to present the properties of the normal distribution, which are essential to the presentation of this work, we must first explain the key average values used in statistics. These are the **mean, median** and **mode** values.

The **mean** value (symbolised as μ) of a specified data set is the arithmetic average, i.e. the result of the the sum of all the values of the data set divided by the number of values present in the set. A general formula of the mean value is shown in 2.2:

$$\mu = \frac{\sum_{i=1}^{N} x_i}{N} \tag{2.2}$$

where N is the number of values present in the data set. To further clarify this, assume the following data set A:

$$A = 1, 2, 3, 5, 6, 8, 11$$

By using the formula from equation 2.2, with $N = 7$, we calculate the mean value from data set A as:

$$\begin{aligned} \mu &= \frac{\sum_{i=1}^{N} x_i}{N} \\ &= \frac{1+2+3+5+6+8+11}{7} \\ &= 5.14 \end{aligned} \tag{2.3}$$

As a result, the mean value of data set A is **5.14**.

The **median** value of a data set is the middle value of the set if we sort it in ascending order. In data set A above, which is already sorted, the median value is **5**. Note that, if the number of the set elements is even, there are two middle numbers. In this case, the median value is the mean of the two middle numbers [8]. Finally, the **mode** value is the number that appears most times in the set. In set A no number appears more than once, so there is no mode.

These three parameters are very important in statistics and probability analyses and are also key properties of the normal, as well as the lognormal distribution, which we will describe later in this work.

The main properties of the normal distribution are:

- The **mean** value μ.
- The **standard deviation** σ.

Since the normal distribution is symmetrical around the mean value, it is worth noting that mean of a normal distribution is equal to its median and mode values. As such, only the mean parameter is used as a main feature of this distribution. The **standard deviation** defines the width of the distribution, i.e. the dispersion around the mean value.

The normal distribution PDF can be described by equation 2.4 below:

$$f(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{1}{2}(\frac{x-\mu}{\sigma})^2} \tag{2.4}$$

where x the value of the examined variable, μ the mean value and σ the standard deviation. If we assume $\mu = 0$ and $\sigma = 1$, an indicative graphical representation of the normal distribution is shown below, creating the well-known "bell-curve":

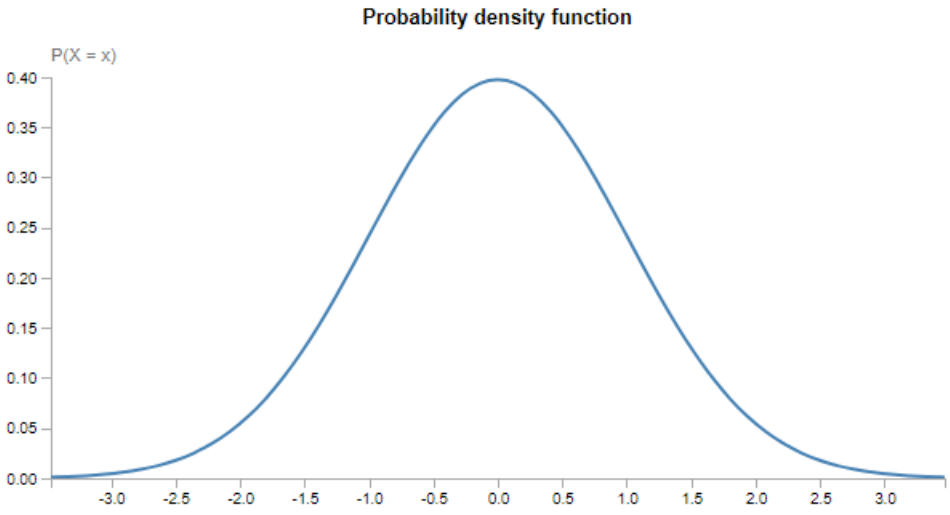


Figure 2.3: Normal Distribution PDF with $\mu = 0$ and $\sigma = 1$

To further test this, in figure 2.4 we generate 5000 random samples, like we did for the uniform distribution above. As expected, the values close to the mean value 0 have a higher frequency of occurrence and the more we deviate from the mean the less probable a value will occur.

Lognormal Distribution

Finally, we proceed to the description of the **lognormal** (or **log-normal**) **distribution**. A random variable x is lognormally distributed when its **natural logarithm** $y = ln(x)$ is normally distributed. As such, a lognormal distribution can be transformed into a normal using logarithmic calculations and vice versa. Apart from the distribution parameters mentioned above, for the lognormal distribution we have

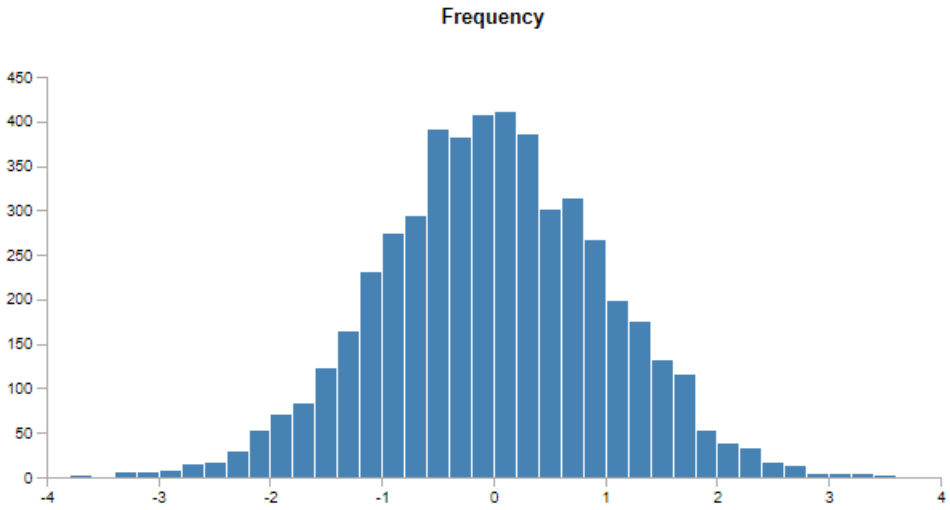


Figure 2.4: Normal Distribution Samples with $\mu = 0$ and $\sigma = 1$

to take into account the **skewness** as well, which describes the degree of symmetry in a distribution. This parameter exists for the normal distribution as well, however, since the normal distribution is always symmetric, its value equals zero. Furthermore, another difference is that, unlike the normal distribution, the mean, median and mode values are not equal in the lognormal distribution. As such, these distributions could be summarised into 3 sub-categories depending on their skewness:

- **The positively or right skewed** distribution, in which the tail is on the right side.
- **The symmetrical distribution**.
- **The negatively or left skewed** distribution, in which the tail is on the left side.

Figure 2.5 shows these 3 sub-categories mentioned above, along with the mean-median-mode relationship for each case:

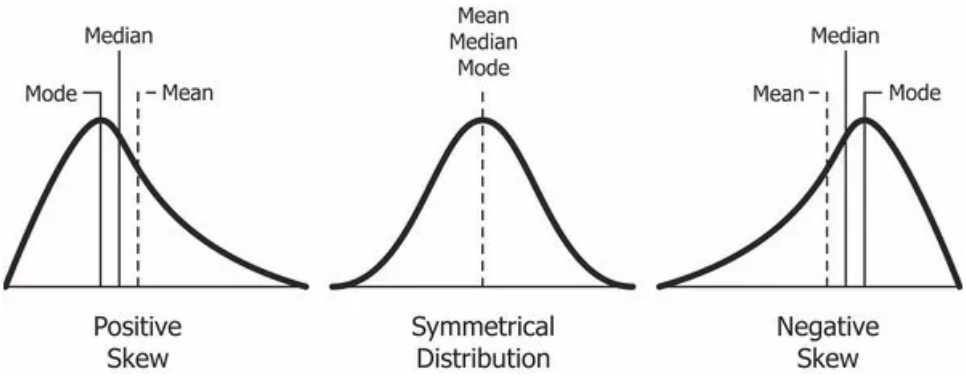


Figure 2.5: Distribution Skewness [9]

The lognormal distribution plays a very important role in engineering, because negative values of specific phenomena are physically impossible [10]. The PDF of the

lognormal distribution can be summarised as:

$$f(x) = \frac{1}{x\sigma\sqrt{2\pi}} e^{-\frac{1}{2}(\frac{\ln x - \mu}{\sigma})^2} \tag{2.5}$$

where:

- μ the **location parameter**.
- σ the **scale parameter**.

It is worth noting that these μ and σ values in equation 2.5 are the mean and standard deviation of the converted normal distribution of the variable $y = ln(x)$, **not** for the lognormal distribution of the variable x. In other words, they define the mean value and standard deviation of the natural logarithm of x. Since it is possible to convert a lognormal distribution to its respective normal and vice versa, we can generate a lognormal distribution by either defining the parameters (μ, σ) of x or $\ln x$. This can be done with the following calculations:

Assume μ_x and σ_x the desired mean and standard deviation of the lognormal distribution of x and μ_y and σ_y the mean and sigma of the normal distribution $y = ln(x)$. These can be calculated using the following equations [11]:

$$\mu_y = ln\left(\frac{\mu_x^2}{\sqrt{\mu_x^2 + \sigma_x^2}}\right) \tag{2.6}$$

$$\sigma_y = \sqrt{ln\left(1 + \frac{\sigma_x^2}{\mu_x^2}\right)} \tag{2.7}$$

Otherwise, if we would rather avoid more complex calculations, μ_y could be calculated after σ_y with the following equation [10]:

$$\mu_y = ln(\mu_x) - \frac{1}{2}\sigma_y^2 \tag{2.8}$$

Finally, similar to the previous examples, we present the graphical representation of the lognormal distribution PDF, assuming $\mu = 0$ and $\sigma = 1$, in figure 2.6 and validate it with the generation of 5000 random samples in figure 2.7.

As we can observe, the generated lognormal distribution converges to zero quickly, leaving multiple data points that rarely occur. In most cases, we assume these samples as the distribution **outliers**, which may affect the quality of the desired result. As such, for the purpose of this work, we use the **interquartile range (IQR)** method, which trims the outliers of the data set, keeping only the data from the 25th to the 75th percentile (50% of the total data) to improve the quality and accuracy of the produced results.

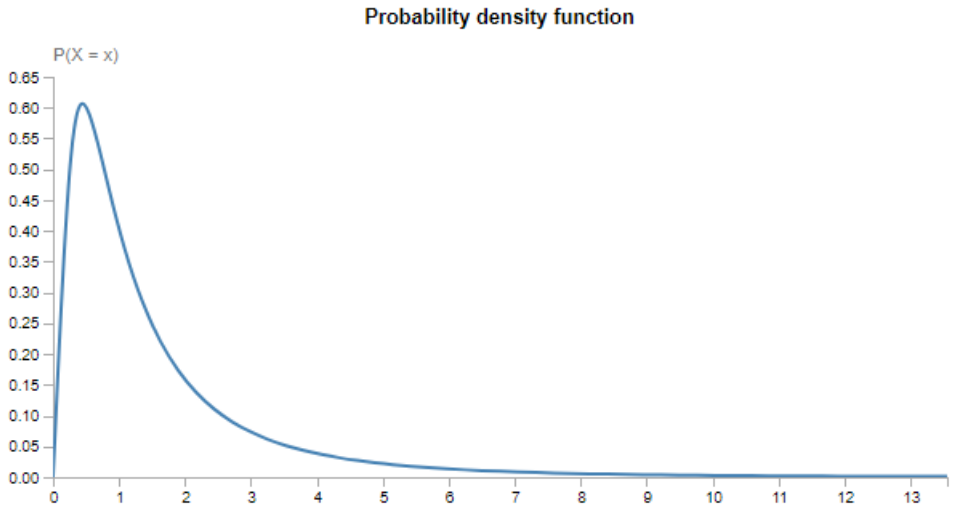


Figure 2.6: Lognormal Distribution PDF with $\mu = 0$ and $\sigma = 1$

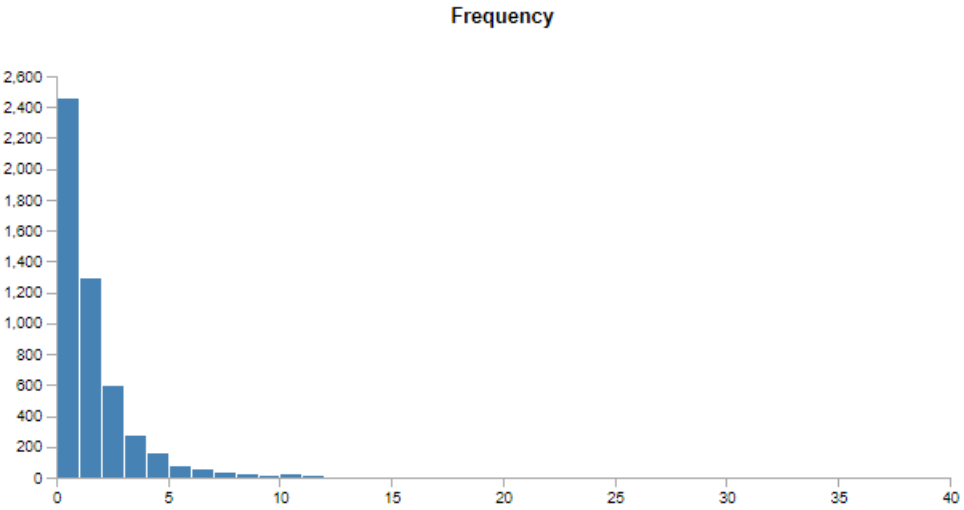


Figure 2.7: Lognormal Distribution Samples with $\mu = 0$ and $\sigma = 1$

2.2 Introduction to STA in Digital Circuits

2.2.1 Static Timing Analysis

Static Timing Analysis (STA) is a technique used to verify the timing of a digital design. Along with timing simulation, they are the two most common approaches for timing verification. However, in contrast to timing simulation, STA computes the expected timing of a design without applying input data values to observe the result. As such, by defining the external environment of the design, including the input clocks, STA can determine whether the design can function at the desired clock frequency [12]. Figure 2.8 shows a basic flowchart of STA.

Static timing analysis is widely recognised and usually preferred over timing simulations in the semiconductor industry for the timing verification of digital designs, due to its primary benefit of quickly checking all the different timing paths of a design. Timing simulation produces timing reports only from the circuit paths

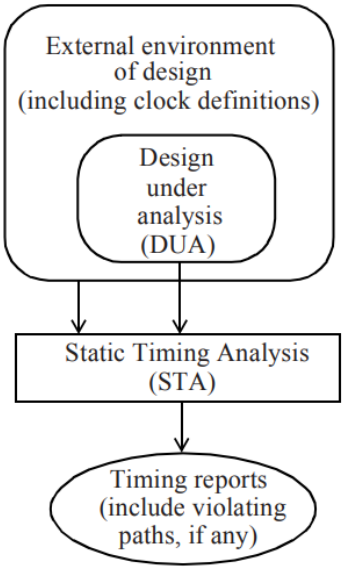


Figure 2.8: Static Timing Analysis [12]

directly affected by the specific data values given as inputs. While STA as a field contains many different concepts, for the purpose of this work we will discuss and present two of the most important ones, which are the **delay** and **slew** values of a digital circuit.

2.2.2 Delay and Slew Propagation

To accurately describe the delay and slew of a circuit, we will consider a simple CMOS **inverter** circuit as an example (figure 2.9).

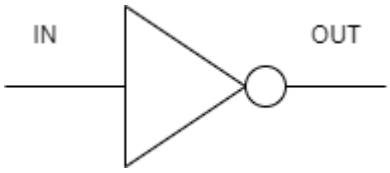


Figure 2.9: The CMOS Inverter

We call **propagation delay** the time between the point where the input signal reaches 50% of its final value and the point where the output signal reaches 50% of its final value. In other words, the propagation delay of a circuit describes the time needed in order for an input change to be evident to the output. As such, it is an essential part of digital circuit design, since inconsistent propagation delays in a design with millions and billions of gates may result in poor functionality [13]. Propagation delay is not a constant value and is determined by the transition time at the input (which can be modified by changing the clock frequency of the design) and the output load of the logic gate, since higher capacity loads require more time to be fully charged. In chapter 5 of this work, we will also examine cells with different loads and compare how the propagation delay is affected.

There are two types of propagation delay, depending on the input and output signal:

- **Fall Delay**
- **Rise Delay**

As their name suggests, fall delay describes the time from 50% of the input rise signal to 50% of the output fall signal, while rise delay describes the opposite. Figure 2.10 shows the rise and fall delays of the CMOS inverter using an approximate input and output waveform:

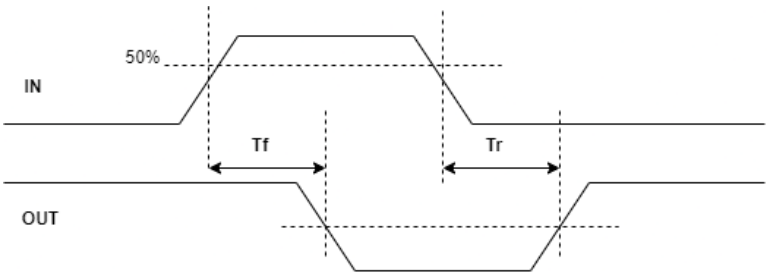


Figure 2.10: Rise and Fall Propagation Delay

For the rest of this thesis, for simplicity, we will refer to propagation delay simply as 'delay'. We will now present the second value needed for the purpose of this work, which is the **waveform slew** or **slew rate** of an output signal.

We define slew rate as the rate of change of the voltage (or current) of an output signal to determine how fast the transition between two levels is. As such, the slew rate is measured as voltage per unit of time and the larger the transition time, the slower the slew and vice versa [12]. Similar to the delay between two signals, the slew rate is also categorised into **rise** and **fall** slew, depending on the signal. Since it is difficult to determine the exact starting and ending points of a signal transition, we choose specific threshold levels as percent of *Vdd* to define the slew rate. For example, a rise slew could be the difference from the time the rising edge reaches 20% of *Vdd* to the time it reaches 80% of *Vdd*. In this work, we measure the slew rates as the time between **the 10% and the 90%** of the signal.

Figure 2.11 presents the measurement of the rise and fall slew rates of a signal waveform:

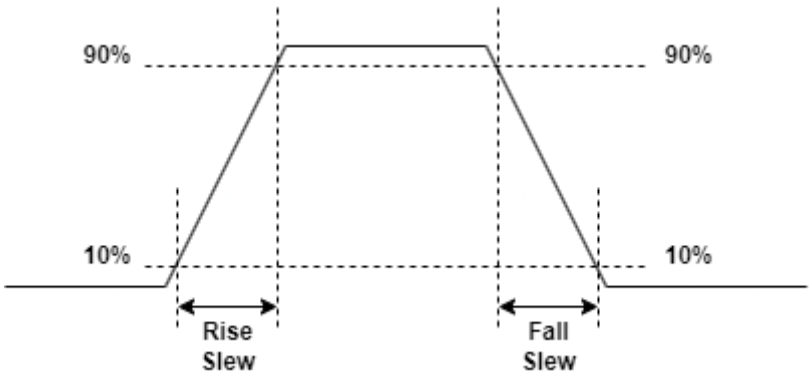


Figure 2.11: Rise and Fall Slew Measurement

The delay and slew measurements of a circuit have a very important role in electronics and digital design, since they, among others, determine the timing and functionality of a design. As such, they are the main terms we examine and experiment with in this work to determine how process variation affects the design. We will briefly review these terms in chapter 4 where we will present our work.

2.3 Introduction to SPICE

SPICE, which stands for **Simulation Program with Integrated Circuit Emphasis**, is an open-source electronic circuit simulator, used for both analog and digital circuits. Its main purpose is to verify the timing and integrity of the circuits and approximately predict their behaviour. Since it is open-source, there are many different versions of SPICE, such as PSPICE, LTSPICE, NGSPICE etc. Some of the SPICE versions use a Graphical User Interface (GUI), while others are terminal based. The main code used to describe and simulate a circuit with SPICE is called a **SPICE deck** or SPICE netlist. We will briefly present the basic features of a SPICE deck used for the purpose of this work, in order to understand how SPICE simulation works.

To add a circuit instance in a SPICE deck, the following must be defined in a single line in that order:

1. Instance name (usually descriptive of its function, e.g. Cout, Cin for a capacitor at the output or input respectively).
2. Circuit nodes to wich the element is connected.
3. The values of the electrical characteristics of the component.

We will go in more detail on the SPICE instances in chapter 4 where we will present the flow of our work. The other most important feature of SPICE is the **analysis method** of the simulation. The most basic types of SPICE simulation analysis are:

- **DC Analysis**, which is used to analyse all the static characteristics of the circuit, i.e. all the DC characteristics.
- **AC Analysis**, which is used to analyse all the frequency-based properties, such as capacitance.
- **Transient Analysis**, which is the time response analysis, i.e. the complete function of the circuit during a given time interval.

To simulate and measure the effects of process variation in our circuits, we use **transient analysis**, since we need to generate the waveforms of each circuit in relation to time.

2.4 Process Variation in Digital Circuits

In semiconductor engineering, we define **process variation** as the variation in the attributes of transistors, e.g. length, thickness, that are present during the

fabrication of the design. These variations not only affect the timing and functionality of the design, but they may also cause malfunctions or even complete failure of the chip. We can separate process variations into two different categories, depending on the number of transistors they affect:

- **Global** variations, which affect all transistors of a design, e.g. temperature variations.
- **Local** variations, which only affect one or a small group of transistors.

The main topic of research on this work is **timing and delay variations**, which, apart from the process variations mentioned above, could also be caused by the following reasons [14]:

1. **Computing errors** in timing analysis, due to either inaccuracies of the device models, either interconnect parasitics which affect the overall timing.
2. **Environmental conditions during device operation**. These kinds of variation are present after the complete manufacturing of the device during its operation. Such are the operating mode, e.g. low power or high performance mode, the temperature and the natural degradation.

While these two factors might play an important role in timing variations , for the purpose of this work, we will focus solely on process timing variations, which occur during the design and fabrication process in transistor-level. To investigate this topic, we applied the technique of derating, which we will briefly explain in the final subsection of this chapter.

2.4.1 Derating

In general, the term **derating** in electronics refers to the concept of a device operating at less than its mentioned maximum capabilities. It could be applied in different parameters of a device, such as voltage or power and its main purpose is to prolong the life of the device. By operating at less than its rated maximum power, the device becomes more resistant to the various environmental stresses, thus reducing its degradation rate and increasing its durability.

Out of the many types of derating, in transistor-level we apply derating at the **transistor parameters**, e.g. length and height, to alter the timing between the circuit components and the design timing in general. The numbers which represent the amount of derating applied to a parameter are called **derate factors**. In our investigation, we generate random derate factors using Monte-Carlo method, apply them to the desired transistor parameters and examine the output timing variations produced by the different outcomes.

Chapter 3

Existing Works

Before we move on to the main presentation of our work, let us examine some of the existing research on the topic of **Statistical Static Timing Analysis (SSTA)**, in which Monte-Carlo simulation in VLSI circuits belongs to. While SSTA has been an area of interest of both academic and industrial research, we will present three academic works which use SSTA to model and investigate circuit power and performance.

3.1 Statistical Static Timing Analysis (SSTA) in the Literature

Statistical Static Timing Analysis (SSTA) refers to the sub-field of STA in which, instead of the conventional deterministic STA algorithms, the timing of the circuits is calculated using probability distributions. As such, the output of this analysis is also a probability distribution of many different outcomes, instead of a single one.

Monte-Carlo simulations are very common in SSTA investigations, with randomly generated distributions to be used as inputs, in order to examine a variety of outcomes. In this subsection, we will briefly present three different academic researches in the field of SSTA, before we move on to the presentation of our investigation flow.

3.1.1 Investigation on Performance, Power & Area using Deterministic and Monte-Carlo Synthesis Flows

To deal with process variation, the industry tries to calibrate the ASIC flow using specific golden silicon data obtained from multiple test chip runs. However, these are mostly applied at the ASIC Back-End flow, i.e. Place & Route, Clock Tree Synthesis, In-Place Optimisation, Sign-Off. As such, to provide further insight into inter-wafer and intra-die process variation as well as improved initial data at the Back-End flow, this first work proposes a deterministic and a Monte-Carlo flow applied at the post-synthesis gate-level [15].

Since the worst-case scenario is extremely rare for most cases, the deterministic flow of this work proposes synthesis at the typical corner accompanied by specific extra actions to close timing in the worst case as well. These extra actions include the worst case (slow corner) timing analysis of the netlist synthesised at the typical process corner and the use of **derate factors** at the critical paths per endpoint, in order for the optimiser to put more effort in these portions of the design to achieve timing closure in the worst case, while also preserving the area benefit of the typical corner synthesis for non-critical paths. The derate factor for each worst case violated path is calculated as the delay ratio between slow and typical corner, as seen in the following equation:

$$Derate = \frac{D_{slow}}{D_{typical}} \tag{3.1}$$

After the derate factor generation, incremental synthesis is performed. Note that timing closure in the worst case may not be achieved after a single run, so multiple iterations might be necessary. As such, the typical netlist attains the current worst case timing conditions and new derate factors are generated. The process is repeated until all worst case timing conditions are met. Figure 3.1 shows a flowchart of this iterative deterministic flow.

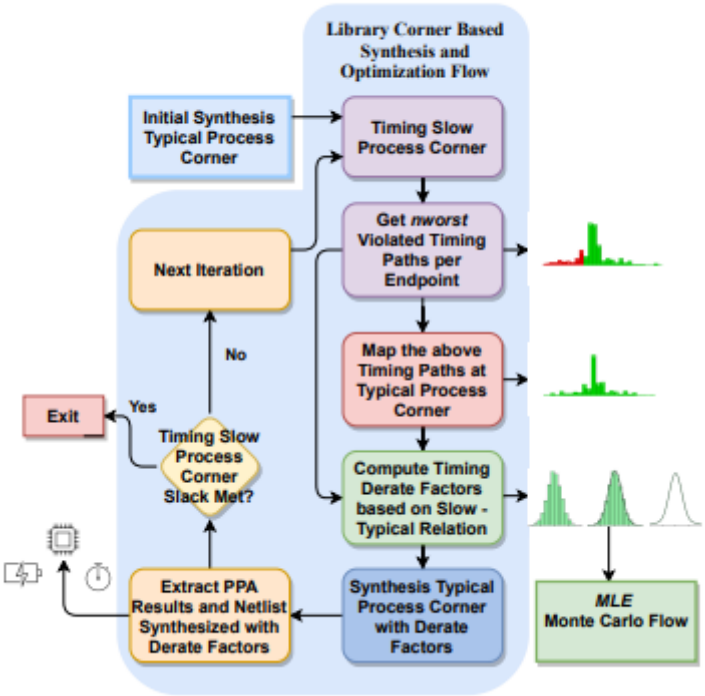


Figure 3.1: Deterministic Library Corner Based Flow [15]

The Monte-Carlo based flow this work also proposes (figure 3.2), uses as basis the derate factors calculated from the previous deterministic flow and, using the **Maximum Likelihood Estimation (MLE) method**, estimates the parameters

of a possible numerical distribution, whose random samples could generate these factors as a result. From the estimated distribution, random derate factors are generated and applied at each component. Then, synthesis at the typical process corner is performed and PPA (Power, Performance and Area) results are produced. The process is repeated by regenerating another set of random derate factors from the same distribution, until the maximum specified number of successful iterations is reached.

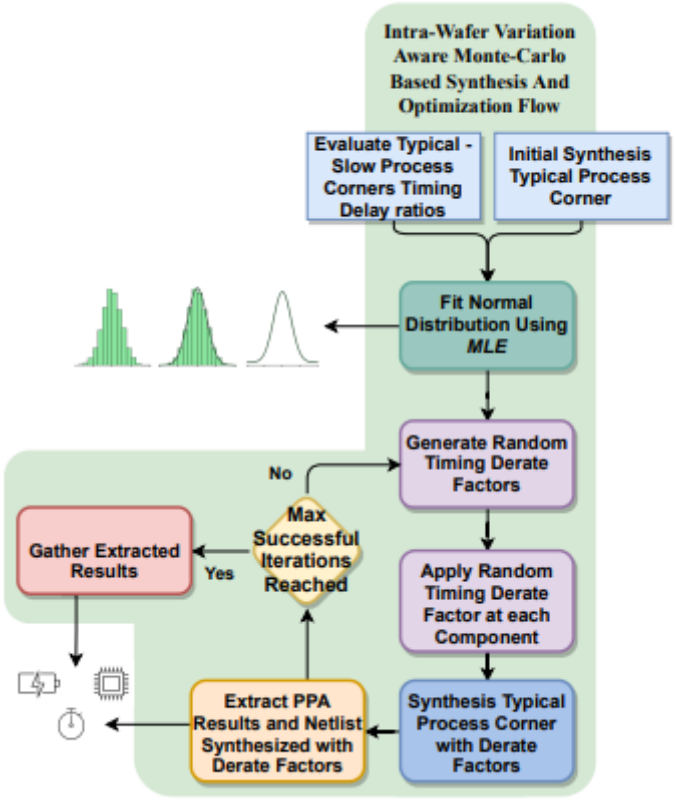


Figure 3.2: Monte-Carlo Based Synthesis and Optimisation Flow [15]

Both of these flows were tested using four open-source designs and produce an average **9.74%** improvement in area and **22.14%** improvement in leakage power in comparison to netlists synthesised at the worst case, while also meeting worst case timing.

3.1.2 A Statistical Performance Simulation Methodology for VLSI Circuits

This second work present a statistical performance simulation (SPS) methodology for VLSI circuits by analysing each smaller circuit block separately and generating the performance distribution for the entire circuit. The main flow consists of the following steps in general [16]:

1. Generate a statistically significant number of SPICE parameters directly from Electrical-Test data.

2. Divide the design into smaller blocks and identify statistically similar and distinct blocks.
3. Construct a model of the full circuit using the response surface methodology (RSM).
4. Generate performance distribution of the full model circuit.

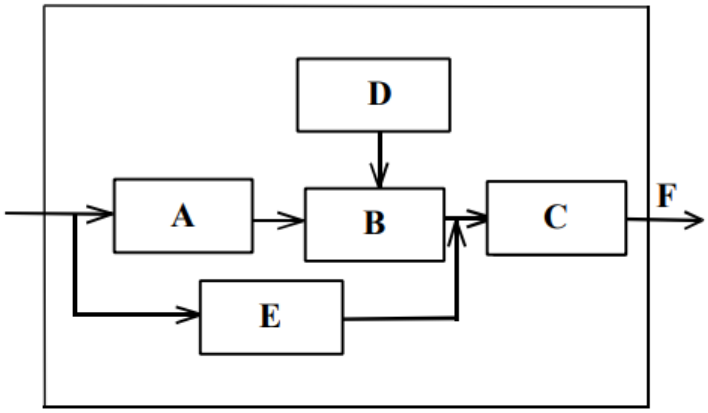


Figure 3.3: Circuit behaviour analysed in blocks [16]

The main idea behind this methodology is to greatly reduce the analysis effort by separating the circuit into statistically distinct sub-blocks. If a block has a statistically similar behaviour to another, then it is excluded from the analysis reducing the necessary simulations to be performed. After all the distinct block performance distributions are generated, as shown in figure 3.4, a full circuit model is created using RSM, which calculates a function relating the block performances to a full circuit performance. This methodology, while less accurate than Monte-Carlo, is computationally more efficient, since identical statistical operations are excluded, saving time [16].

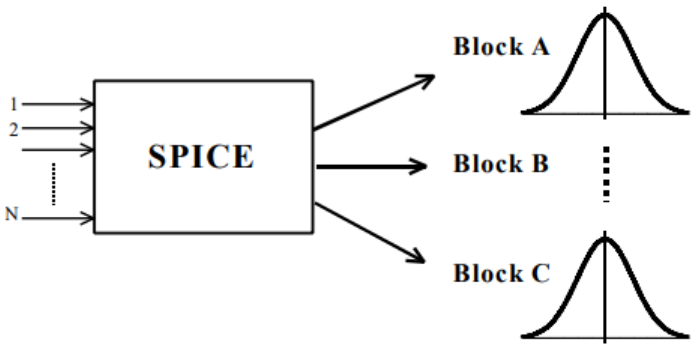


Figure 3.4: Block Performance Distribution Generation [16]

The paper provides more information on RSM and related algorithms used, as well as experimental results and observations.

3.1.3 Modelling Circuit Performance Variations due to Statistical Variability: Monte Carlo Static Timing Analysis

The final work, which we will present here, investigates the impact of random intra-die statistical variations on digital circuit timing and power consumption. It compares traditional corner-based STA with Monte-Carlo SPICE simulations and their proposed method of **Monte-Carlo Static Timing Analysis (MCSTA)**, all tested on a one bit full adder [17].

Monte-Carlo SPICE Simulations

For this method, RandomSpice was used, a tool which acts as a circuit simulator, while also providing statistical analysis support. RandomSpice replaces all MOSFET model instances within a SPICE netlist with randomly picked BSIM instances from a specific statistical library [17]. The threshold voltages of each transistor was randomly generated from a Gaussian distribution, thus injecting variations into the design. The investigation was performed on seven levels of threshold voltage variation (which is represented by the standard deviation σ_{VT} of the Gaussian distribution), from 10% to 50% and 10000 SPICE netlists were generated.

Process Corner Analysis

As mentioned above, corner analysis refers to applying STA while setting process and environmental parameters at extreme cases, e.g. worst case scenario (slow corner), typical corner etc. To apply statistical variations, multiple standard cell libraries were generated at $\pm 3\sigma_{VT}$, where σ_{VT} is the standard deviation of the threshold voltage Gaussian distribution, calculated from the Monte-Carlo SPICE simulations, while also applying the same simulation input. Since these extreme cases have a significantly low possibility to occur, corner based STA usually produces a relatively pessimistic result [17].

Monte Carlo Static Timing Analysis

This proposed method combines the accuracy of Monte-Carlo SPICE simulations with the simplicity and quickness of STA. RandomSpice generates multiple randomised netlists of each standard cell in order to create an equivalent standard cell library which includes the statistical differences between transistors [17]. Then, STA is applied to all of these different randomised netlists in order to produce a timing and power consumption distribution. Like the two previous methodologies, the same input was used and at the end of the work, comparison of all three methods was presented.

3.2 Thesis' Used Tools

3.2.1 Cadence Spectre Circuit Simulator

For the experiments of our work, we used **Cadence Spectre Circuit Simulator**. Spectre is owned and distributed by Cadence Design Systems and provides all the basic SPICE features and analyses, as well as SPICE language support. However,

Spectre also includes many improvements over other SPICE tools, such as improved capacity, accuracy and speed [18]. Apart from the SPICE language support, Spectre has its own language as well. In this work, we use SPICE netlists for circuit descriptions and measurements, while including Spectre language blocks to perform the Monte-Carlo simulations. The target language can be changed any time inside a netlist with the *simulator lang* command. We will provide further information regarding the used Spectre code blocks in the next chapter, where we present our transistor level Monte-Carlo flow.

Chapter 4

Monte-Carlo Simulations in Digital Circuits

After presenting the necessary background knowledge, in this chapter, we proceed to the presentation of our work. To start with, a sample flowchart of the transistor level Monte-Carlo flow is shown, describing in detail each of the steps in the process. Each subsection contains either the description of the input files used, or the actions to be executed by the user.

The Transistor Level Monte-Carlo Flow is executed by a simple bash script, which will be described below, while Python was used for the SPICE deck generation, as well as the collection and visualisation of the results using the *matplotlib* library. Though other programming languages could also be used for this project, Python was selected, due to its simplicity and direct interactivity with the user.

4.1 Transistor Level Monte-Carlo Flow

In 4.1 the basic flowchart of the Transistor Level Monte-Carlo Flow is presented. The purple block corresponds to the main bash script to be used directly by the user. All the other steps are executed automatically from the script using the input parameters given by the user. The Python scripts used are shown in green colour, while the Spectre execution is shown in red colour.

The flow takes as inputs the following parameters in this **specific order**:

1. *"Source"* command and the name of the script, i.e. *source run_flow.sh*.
2. The number of Monte-Carlo iterations performed.
3. The number of chain components, i.e. buffers or inverters.
4. Input signal type, i.e. rise or fall.
5. Buffer or inverter chain declaration, i.e. *buf* of *inv*.
6. The specific libcell or libcells to be used from the library, ex. *BUFx12_ASAP7_75t_R*.

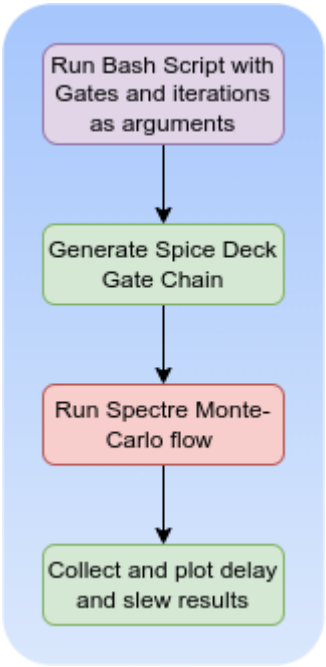


Figure 4.1: Automated Spectre Monte-Carlo Flow

4.1.1 Models Used

For the purpose of this work, a modified version of the ASAP7 7-nm finFET predictive process design kit was selected, with a maximum voltage threshold of 0.7 V. This specific library was developed by Arizona State University (ASU) in collaboration with ARM and is open-source, so it is easy to use and modify according to one's objective. The PDK, cell libraries and SPICE models are available for download on GitHub and are regularly updated [19].

4.1.2 Parameters Under Variation

Normally, the transistor length would be an ideal parameter for variation. In this case however, as shown in 4.2, the transistor length is much shorter than the transistor width, so process variations in the length are negligible. Thus, the main parameter under testing is the width. Note that in the transistor models file, the width (*wfin* in 4.2) is not a standalone parameter, but a combination of transistor height and thickness, i.e. *hfin* and *tfin* respectively. In order to perform the necessary Monte-Carlo simulations, derate factors must be applied to the aforementioned parameters, as explained previously in section 2.4.1.

So, to apply these derate factors, two spectre language Monte-Carlo blocks were added to the models file, one for the NMOS transistor and one for the PMOS. Assuming **x** derate factor for hfin and **y** derate factor for tfin, we generate each one of them randomly from a specific distribution and the total number of samples equals the number of Monte-Carlo iterations to be performed. Finally, we multiply the default parameters for height and thickness with the generated derate factors.

Below, we present the Spectre code block for the derate factor generation. Note that the following code is **not** used to perform the simulations. This part is included

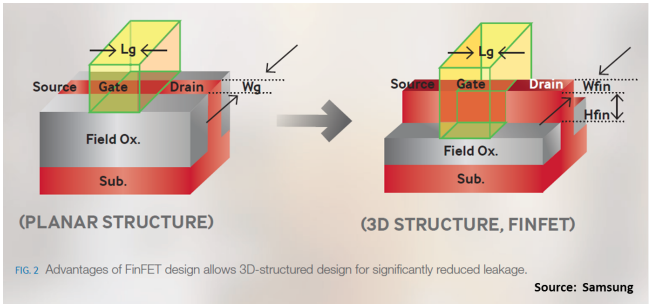


Figure 4.2: FinFET Channel Length [20]

in the SPICE deck of each design and will be explained in the next subsections.

```
simulator lang=spectre
        parameters x = 1
        statistics {
                process {
                        vary x dist=gauss std=0.2 percent=no
                }
                truncate tr=5
        }

        Vxparam (Vxnode 0) bsource v=x

        parameters y = 1
        statistics {
                process {
                        vary y dist=gauss std=0.2 percent=no
                }
                truncate tr=5
        }

        Vyparam (Vynode 0) bsource v=y
simulator lang=spice
```

Table 4.1: Derate Factor Generation

Before each *statistics* block, the default values of the derate factors are set, which also act as the mean values of each numerical distribution from which the numbers are randomly generated. In this case, as presented in each *statistics* block and in figure 4.3, the normal (gaussian) distribution is used, with a mean value of 1 and a standard deviation of 0.2. Finally, a truncate factor is used to set the range of valid generated values. Each result which is not within the appropriate range, is rejected and regenerated until a valid value is given.

So, assuming truncate factor **tr**, the range limits are calculated as:

$$limits = mean \pm (tr \cdot std) \tag{4.1}$$

From equation 4.1, we can determine the range of accepted factors. In this case, the truncate factor is 5, so the range limits are 0 and 2 respectively. As such, any value lower than 0 or higher than 2 will be considered invalid and regenerated

immidiately until a valid one appears. Constraining the results and truncating invalid values is obligatory, since there cannot be negative derate factors and the simulations will result in failure.

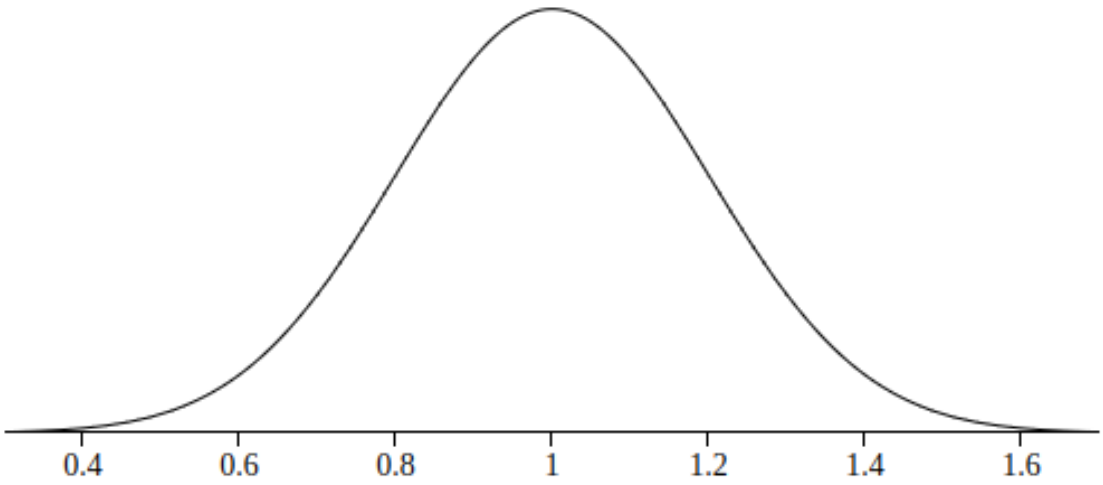


Figure 4.3: The Normal Distribution used

Finally, the *Vxsource* and *Vysource* commands in the code are solely used for the visualisation of the generated derate factors to ensure their validity. They do not impact the functionality of the flow in any way.

4.1.3 Designs Under Investigation

For the purposes of this work, the designs we used are buffer and inverter chains, using the library cells presented above. As mentioned at the beginning of the chapter, the user executes the bash script and sets the appropriate parameters as input from the command line. The flow executes the Python script for the SPICE deck generation, so it is not required for the user to write it manually. Then, Cadence Spectre Simulator is executed using the generated SPICE deck. For each step of the chain, delay and slew are measured, as shown in 4.4, 4.6 and 4.7. After Spectre completes its execution, the next Python script is executed, which takes the delay and slew measurements of the last buffer or inverter of the chain and plots them in a histogram, in order to determine how the results are distributed. The two Python scripts will be described in more detail in the sections 4.1.4 and 4.1.5 below.

The following figures present the designs that we tested along with the measured values, depending on the input signal.

To start with, we will present the buffer chain, which is also the simplest one. As we can see in 4.4, if the input signal is rise, we measure and store the following values:

- **Rise Delay** as the time interval between the 50% of the input signal and the 50% of the output signal.
- **Rise Slew** as the time interval from 10% to 90% of the output signal.

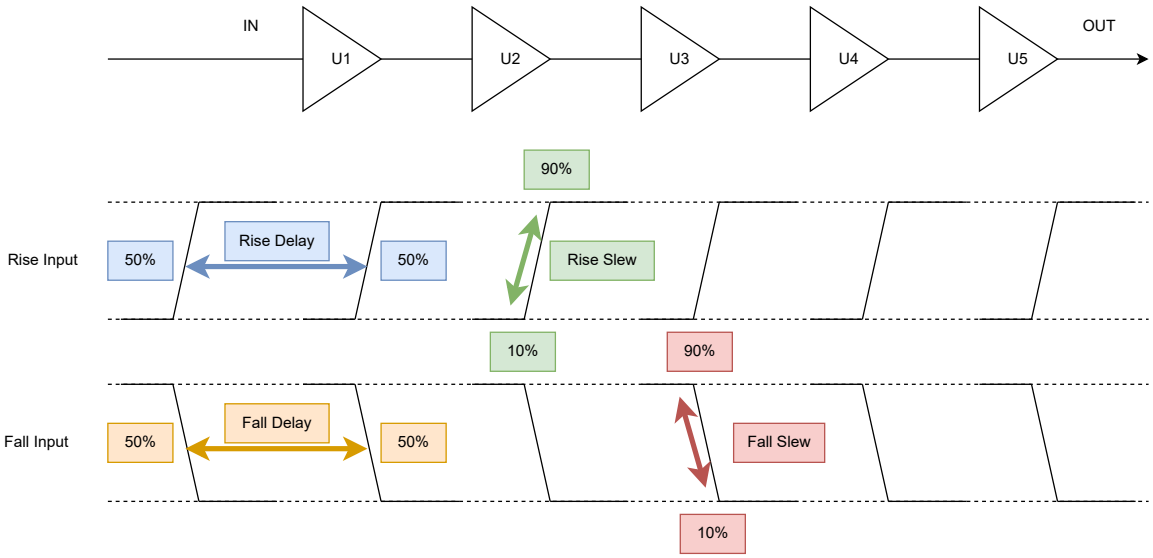


Figure 4.4: Buffer Chain

Likewise, if the input signal is fall, we measure the following values:

- **Fall Delay** as the time interval between the 50% of the input signal and the 50% of the output signal.
- **Fall Slew** as the time interval from 90% to 10% of the output signal.

Note that in the buffer chain, since the input signal edge does not change from low to high or vice versa, we can easily determine the last delay and slew values -for plotting purposes- from the number of chain components given as input. However, things are more complicated when it comes to inverter chains.

As the name suggests, an inverter shifts the input signal. So, for example, if the input signal is rise, the output produced will be fall, as shown in 4.5, where the behaviour of an inverter with a changing input is presented. When the input (red waveform) is low, the output (yellow waveform) becomes high and vice versa. This affects the measurements of the delay and slew as well, since the final output varies depending on both the input signal and the number of the chain components.

To be more specific, the inverter chains are separated into two categories, presented in figures 4.6 and 4.7 respectively. As such, we take the following measurements for each case:

1. If the inverter chain contains **odd** number of inverters (figure 4.6):
   (a) If input signal is **rise**:
   - **Propagation High-to-Low (PHL) Delay** as the time interval between 50% of the rise input signal and 50% of the fall output signal.
   - **Fall Slew** as the time interval from 90% to 10% of the fall output signal.

   (b) If input signal is **fall**:
   - **Propagation Low-to-High (PLH) Delay** as the time interval between

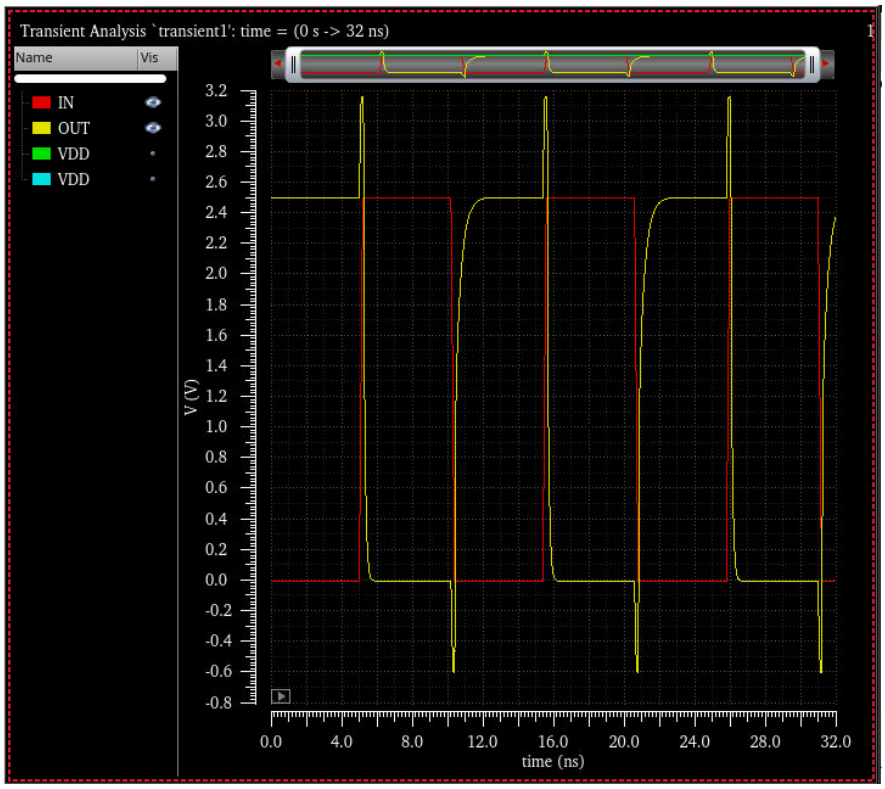


Figure 4.5: Single Inverter Input and Output

50% of the fall input signal and 50% of the rise output signal.

- **Rise Slew** as the time interval from 10% to 90% of the rise output signal.

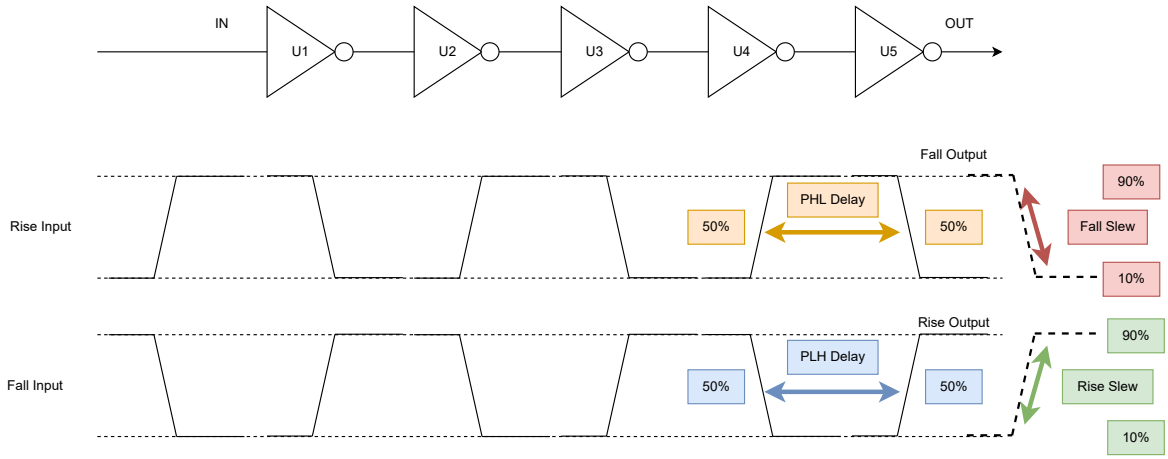


Figure 4.6: Inverter Chain

2. If the inverter chain contains **even** number of inverters (figure 4.7):

   (a) If input signal is **rise**:

   - **Propagation Low-to-High (PLH) Delay** as the time interval between 50% of the fall input signal and 50% of the rise output signal.
   - **Rise Slew** as the time interval from 10% to 90% of the rise output

signal.

(b) If input signal is **fall**:

- **Propagation High-to-Low (PHL) Delay** as the time interval between 50% of the rise input signal and 50% of the fall output signal.
- **Fall Slew** as the time interval from 90% to 10% of the fall output signal.

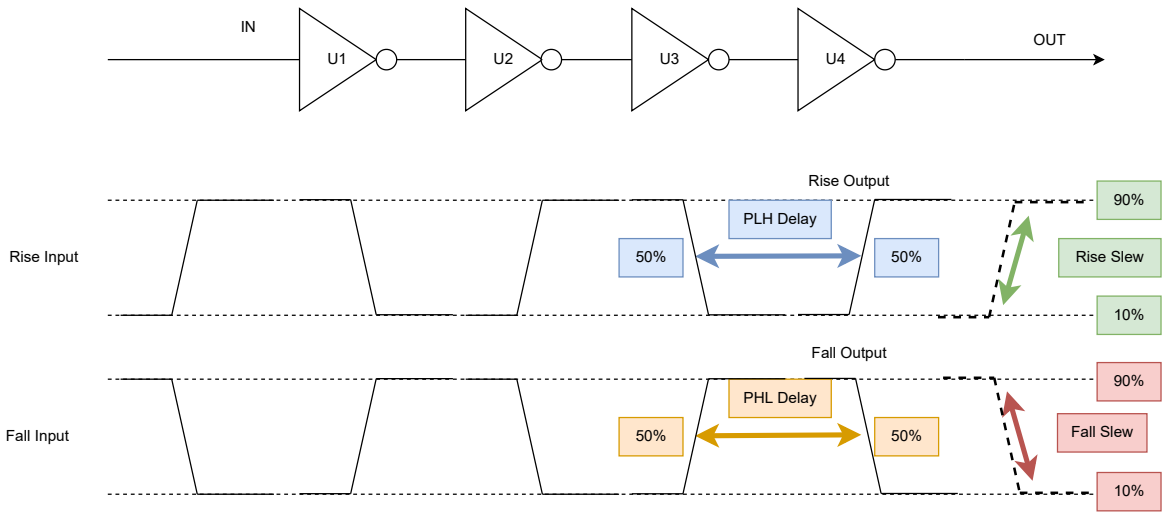


Figure 4.7: Inverter Chain with Even Number of Inverters

So, to summarise, the above measurements are performed in every component of the chain for each Monte-Carlo run. At the end of the simulation, Spectre saves all the values in the *design.measure* file and shows the minimum and maximum values, as well as the mean value at the terminal. Out of these results, only the last delay and slew values are presented in histogram form. More details about the results of the experiments will be described in chapter 5 below.

4.1.4 SPICE Deck Generation

The first out of the two Python scripts that the flow utilises is responsible for the automated generation of the component chain SPICE deck. This is mostly done by a string that stores all the necessary information and it is printed as an output and redirected to a new file at the end of the script. The inputs of the script are the number of Monte-Carlo iterations, the number of chain components and the target libcell or libcells. The script function is almost identical for all design cases presented in 4.1.3, with some slight differences concerning the measurements of the inverters. The rise input buffer chain case will be used as the main reference point for explaining the script, while noting any necessary alterations for the other cases.

We will now move on to the basic description of the script. Firstly, a string constant named *INITIALBLOCK* is set to write the basic SPICE information needed, i.e. the inclusion of the transistor model files to be used, the ground and power supply (GND and VDD respectively), the initial diver buffer or inverter with the input

signal and the initial conditions. In the SPICE language each component is written in this format:

[NAME] [VSS] [VDD] [IN] [OUT] [LIBCELL]

So for example, by using this format, the first component is declared as:

XU1 VSS VDD U1:A U1:Q BUFx2_ASAP7_75t_R

The rest of the process is automated in a *for* loop and varies depending on the number of chain components. Since the circuit is a simple chain, the output of the previous component is the input of the next, so we use the loop index (named *uindex*) to determine each component name, input and output wire. The loop code can be seen explicitly in table 4.2 below:

```
for uindex in range(2, lastcomponent):
        fullgatename = "XU" + str(uindex)
        inputnet = "U" + str(uindex - 1) + ":Q"
        outputnet = "U" + str(uindex) + ":Q"

        instanceline = fullgatename + " VSS" + " VDD " + inputnet
        + " " + outputnet + " " + gates[uindex%gateslen] + "\n"

        # add each instance line to the final spice deck
        spicedeck += instanceline
```

Table 4.2: Chain Component Declaration in Python Script

It is worth mentioning that the last parameter *gates* of the *instanceline*, which is the name of the libcell or libcells to be used in the chain, is an array of strings containing the different libcell names. In case of multiple libcells, a round robin policy is used, so the libcells are used consecutively depending on their order stored in the array. As such, the modulo calculated by divining the current chain *uindex* by the total count of libcells given (*gates* array length) is the index pointing at the libcell for each iteration, which creates a round robin order and returns to the first array slot after the last one is used. For the purpose of this work, only a **single** libcell type was used for each different experiment, however, it was necessary to mention this feature, since it might be useful for future work and experiments.

After adding all the components instances in the output SPICE deck string, we proceed to the delay and slew measurements. Similar to the *INITIALBLOCK* section, the first buffer or inverter measurement is declared with a constant string which is added to the string output. As mentioned in 2.3, to calculate the appropriate values, the SPICE *".MEAS"* command is used. In the case of the rise input buffer chain, the delay and slew calculations of the first buffer are written as:

- **.MEAS TRAN U1:Q:RISE_SLEW trig V(U1:Q) VAL=0.070000 RISE=1 targ V(U1:Q) VAL=0.630000 RISE=1**
- **.MEAS TRAN U1:Q:RISE_DELAY trig V(U1:A) VAL=0.350000 RISE=1 targ V(U1:Q) VAL=0.350000 RISE=1**

In other words, for the rise slew, the starting point, i.e. **trigger**, for the measurement is the first time the **rise** signal of **U1:Q** reaches a voltage of 0.07 V (10% of VDD) and

the ending point, i.e. **target**, is the first time the same signal reaches a voltage of 0.63 V (90% of VDD). Likewise, the rise delay is measured from the point where the input **U1:A** reaches 0.35 V (50% of VDD) until the point where the output **U1:Q** reaches the same voltage value. For the fall input buffer chain, the delay and slew are calculated using the same logic, but with reversed trigger and target values for the slew (from 90% to 10%) and with **FALL=1** indication, meaning that the measurements are to be performed on the falling slope of the signal instead of the rising.

After the first measurement, the others are automated in a *for* loop presented below, which functions in a similar way with the loop for the component declaration above (table 4.2). After each line is generated, it is added to the output string.

```
for uindex in range(2, lastcomponent):
  inputnet = "U" + str(uindex - 1) + ":Q"
  outputnet = "U" + str(uindex) + ":Q"

  measureslew=".MEAS TRAN "+outputnet+":RISE_SLEW trig V("+outputnet+")
      VAL=0.070000 RISE=1 targ V("+outputnet+") VAL=0.630000 RISE=1"

  spicedeck += measureslew + "\n"

  # if uindex == chaincomponents:
  measuredelay=".MEAS TRAN "+outputnet+":RISE_DELAY trig V("+inputnet+")
      VAL=0.350000 RISE=1 targ V("+outputnet+") VAL=0.350000 RISE=1\n"

  spicedeck += measuredelay + "\n"
```

Table 4.3: Rise Delay and Rise Slew Measurements For Every Buffer

The fall input buffer chain values are measured in the same way. However, there is an important difference to the measurements when it comes to inverter chains. Like we described in section 4.1.3 above, the inverter chains are separated into two categories, depending on whether we have even or odd number of inverters and these two categories are also separated into two different subcategories, according to the input signal. Since the input slope is reversed after each inverter instance, the quantities to be calculated are different after each component. For example, assuming we have a rise input signal, the first inverter measurements will be **PHL_DELAY** and **FALL_SLEW**, the second inverter measurements will be **PLH_DELAY** and **RISE_SLEW** etc. In general, this pattern could be implemented as (for rise input):

- We measure **PLH_DELAY** and **RISE_SLEW** for **even** numbered inverters.
- We measure **PHL_DELAY** and **FALL_SLEW** for **odd** numbered inverters.

Because of this, the above loop is different for the inverter chains. In 4.4 the measurement code block for the inverter chains with rise input is presented. For the chains with fall input, the pattern is the opposite, i.e. for **even** numbered inverters we measure **PHL_DELAY** and **FALL_SLEW** and for **odd** numbered inverter we measure **PLH_DELAY** and **RISE_SLEW**. Since the code block for this case is almost identical, it will be omitted for simplicity reasons. Also note that, in this part of the flow, the number of inverters (even or odd) does not affect the implementation. This difference only concerns the results collection.

```
# for even numbered inverters, measure low-high delay and rise slew
# for odd numbered inverters, measure high-low delay and fall slew
for uindex in range(2, lastcomponent):
inputnet = "U" + str(uindex - 1) + ":Q"
outputnet = "U" + str(uindex) + ":Q"

if (uindex % 2 == 0):
 measureslew = ".MEAS TRAN "+outputnet+":RISE_SLEW trig V("+outputnet+")
 VAL=0.070000 RISE=1 targ V(" + outputnet + ") VAL=0.630000 RISE=1"

 spicedeck += measureslew + "\n"

 measuredelay = ".MEAS TRAN "+outputnet+":PLH_DELAY trig V("+inputnet+")
 VAL=0.350000 FALL=1 targ V(" + outputnet + ") VAL=0.350000 RISE=1\n"

 spicedeck += measuredelay + "\n"
else:
 measureslew = ".MEAS TRAN "+outputnet+":FALL_SLEW trig V("+outputnet+")
 VAL=0.630000 FALL=1 targ V(" + outputnet + ") VAL=0.070000 FALL=1"

 spicedeck += measureslew + "\n"

 measuredelay = ".MEAS TRAN "+outputnet+":PHL_DELAY trig V("+inputnet+")
 VAL=0.350000 RISE=1 targ V(" + outputnet + ") VAL=0.350000 FALL=1\n"

 spicedeck += measuredelay + "\n"
```

Table 4.4: Rise Delay and Rise Slew Measurements For Every Inverter

Following this loop, we proceed to the Monte-Carlo simulation section of the deck. Since we use Spectre as the simulator, we switch the language to the native Spectre language, as we mentioned in 3.2, for this part. This is mandatory, due to the fact that each circuit simulator program has its own unique Monte-Carlo simulation and while Spectre is compatible with the SPICE language, it cannot recognise all the different functions that are unique to different tools. In the Spectre language, a Monte-Carlo simulation is declared as follows:

```
simulator lang=spectre

    mc1 montecarlo variations=all seed=1234 numruns=[no. of runs] {
        sw1 sweep param=temp values=[27] {
                tran1 tran start=0 stop=5n step=0.001p
        }
    }

    simulatorOptions options save=none

simulator lang=spice
```

Table 4.5: Spectre Language Monte-Carlo Runs Declaration

By observing the Spectre code in table 4.5, in order for the tool to run Monte-Carlo simulations correctly, the following conditions must apply:

1. A *montecarlo* block, named *mc1* in this case, must be declared, along with the

appropriate parameters:

(a) **variations** to specify which type of variations to apply: process, mismatch or both. While mostly process variations concern us in this work, for these specific experiments, there is no important difference, so the parameter is set to *all*.

(b) **seed** to determine the seed value of the pseudo-random derate factor generation. In this case, the default 1234 value is set.

(c) **numruns** for the number of Monte-Carlo runs to be executed. In this flow, this number is taken as an input from the command line, so it is left to the user's discretion. However, for the purpose of this work, all the experiments performed **2000** runs.

2. Inside the *montecarlo* block, a *sweep* block is obligatory, mostly to set the different temperatures for the experiments. However, since sweep analyses go beyond the scope of this thesis, all the values remain unchanged at their default values, including the temperature at 27°C.

3. Finally, inside the *sweep* block, there is a *tran* command, which is similar to SPICE transient analysis. As seen in the code above, the paremeters set are the starting and ending points and the step, which is the time interval between measurement repetitions. The values set in the code sample are the same for all the experiments.

Important Note: For the Monte-Carlo simulations to take effect, one or more ***statistics*** blocks, which are responsible for the component variations, must be declared in the file containing the parameters to be varied, as presented in section 4.1.2. The simulator will read those blocks and generate the random derate factors for each run. In any other case, the Monte-Carlo block will not have any parameters to variate and the simulator will produce a warning.

Finally, after the Spectre Monte-Carlo block is inserted into the output string, the SPICE deck is complete and is written using the Python *print* command to a newly created *".sp"* file. Then, the bash flow executes Spectre simulator using the following command:

spectre -64 generated_spice_deck.sp

Spectre performs all the necessary simulations and produces the requested measurements after each run. At the end of last Monte-Carlo run, the minimum, maximum and mean values of each result is presented. More about the results will be presented in chapter 5.

4.1.5 Results Collection

At last, we have the second Python script, which is also the final step of the Spectre Monte-Carlo flow and is used for the collection and visualisation of the results. After Spectre finishes all the simulations, it produces a *.measure* file which contains all the results from each run. For example, in this case, since we want to measure the delay and slew of each buffer or inverter, the file will contain all these values for each Monte-Carlo iteration. This file will be used as input for the

script and, by using Python libraries ***matplotlib*** and ***numpy*** we will visualise in a histogram the desired results, which in this case are the delay and slew of the **last** buffer or inverter. Although with the following explanation of the script, it would be simple to plot the delay and slew values of all the components, we omit them since the results would not be that different and thus the more output files are redundant.

The main purpose of the result visualisation is to investigate how the desired output values are distributed for the different iterations by taking into consideration the **normally distributed** derate factors from the input. To be more specific, in the VLSI field, this is useful to determine in which values the delay and slew of a circuit converge, so that the designer will constrain the chip accordingly. This result examination is done by counting how many times an output is repeated between Monte-Carlo iterations and by plotting these repetitions for each different number. So, in detail, the designated results are presented in a Cartesian coordinate system like this:

- **X-axis** contains all the different output values produced from the *.measure* file, i.e. all the delay and slew values of the last component.
- **Y-axis** contains how many times each respective value of the x-axis is generated.

As a result, the generated graph will take the form of a **numerical distribution**, which is a logical conclusion, since the input derate factors are normally distributed. Also, in order to generate the results correctly, a simplification of the delay and slew numbers must be done: Spectre produces each output with high accuracy, thus containing many decimal point digits. If we take all the numbers unmodified as Spectre generates them, the plot or histogram will be unreadable, since, if we take into account all the digits, no number is repeated more than once. However, if we round all the numbers to the same amount of **significant digits**, we could create the histogram and determine its distribution, without affecting the quality of the results, since the latter digits in a floating point number do not make much of a difference in cases like this. For the experiments of this work, the parameter *SIGNIFICANTDIGITS* for the buffer chains was set to 13 and for the inverter chains to 14, except where noted otherwise.

With that in mind, we proceed to the explanation of the script. Its functionality could be summarised in the following steps:

1. Two integer arrays are set, for delays and slews respectively.
2. Two string for the names of the last component measurements are set.
3. The *.measure* file is opened in read mode.
4. In a loop, the file is read line by line and two checks are performed:
    - If a component delay name is found, the next line is read using the *readline* command and the number is rounded and appended to the delay array.
    - Same process, but for slew.
5. Using the *unique* function of the ***numpy*** library, the frequency of occurrence

of each unique delay and slew value is calculated, i.e. how many times a number is repeated in the iterations. Each of these frequencies is stored in the *unique* array, the length of which will be used as the total count of bins in the histogram.

6. Finally, using the *hist* function of the ***matplotlib*** library, the histograms presenting the distribution for the delay and slew are created and stored.

While the process is almost the same for the inverter chains, there is a slight difference. As we mentioned in 4.1.3, the last component measurements are different depending on the total count of inverters (even or odd) and the input. So, according to these cases presented above, we set a flag variable *mode* to determine which values to look for in the Spectre output file. In the code, all the cases can be summarised like this:

```
# get last component delay and slew, depending on the input signal
# and the number of chain components
if ((inputsignal == "rise" and inverters % 2 != 0) or
    (inputsignal == "fall" and inverters % 2 == 0)):

        lastcomponentdelay = "u" + chaincomponents + ":q:phl_delay"
        lastcomponentslew = "u" + chaincomponents + ":q:fall_slew"
        mode = 1

elif ((inputsignal == "rise" and inverters % 2 == 0) or
    (inputsignal == "fall" and inverters % 2 != 0)):

        lastcomponentdelay = "u" + chaincomponents + ":q:plh_delay"
        lastcomponentslew = "u" + chaincomponents + ":q:rise_slew"
        mode = 2
```

Table 4.6: Last Component Measurement Cases for Inverter Chains

After the collection and rounding of the numbers in the arrays, a *mode* check is performed, in order to create the correct histograms. Apart from these slight alterations, the rest of the results collection process stays the same. The script then stores the histograms locally in **png** format and the execution of the flow is terminated.

This concludes the explanation of the Transistor-level Monte-Carlo flow. In the next chapter, the results of our experiments with the flow will be presented and explained.

Chapter 5

Experimental Results

As mentioned in the previous chapter, when Spectre finishes all the Monte-Carlo runs, the terminal shows a summary of the results stored in the *.measure* file. At the end of each Spectre run, we can see the minimum, maximum and mean values of every variable measured. The sknewness of the distribution of each value is also shown. It is important to observe this summary of the results before moving on to the histogram examination, since from these values we can determine if an error in the calculations has occurred. If all these values (min, max, mean, skewness) are the same for each different entry, then each random derate factor generated was the same. This indicates that the derate factor generation was not successful, because by applying the same factor in every run, there is no process variation. In any other case, we can assume that the flow was completed successfully and examine the results produced by the last step of the flow.

The *.measure* file contains all the calculated delay and slew values from the Spectre runs. As explained in section 4.1.5, only the last chain component delay and slew results are presented graphically for simplicity reasons and to avoid redundancy. The two key factors of the experiments are the number of components and the **load drive** of the selected libcell, i.e. BUFx2, BUFx4 etc, with the BUFx4 driving higher load than the BUFx2. By experimenting with these two design properties, we can examine the behaviour of the delay and slew in many different situations. For the buffer chains, we used chains of 4 and 20 components respectively and for the inverter chains, the same count was used along with the inclusion of 21-inverter chains, to examine the odd-numbered chains as well.

As for the load drive of the libcells, the concept was to try three different cases: low, middle and high. So, the following libcells were selected:

- Buffers:
  - BUFx2 (lowest)
  - BUFx10
  - BUFx24 (highest)
- Inverters:
  - INVx2 (lowest)

- INVx8
- INVx13 (highest)

In the chapter sections, all the results histograms will be presented and compared. Section 5.1 contains the buffer chain experimental results, while section 5.2 contains the inverter chain experimental results. After the explanations of all the graphs, a conclusive summary of all the results will be included for possible further expansion of this research work.

5.1 Buffer Chains

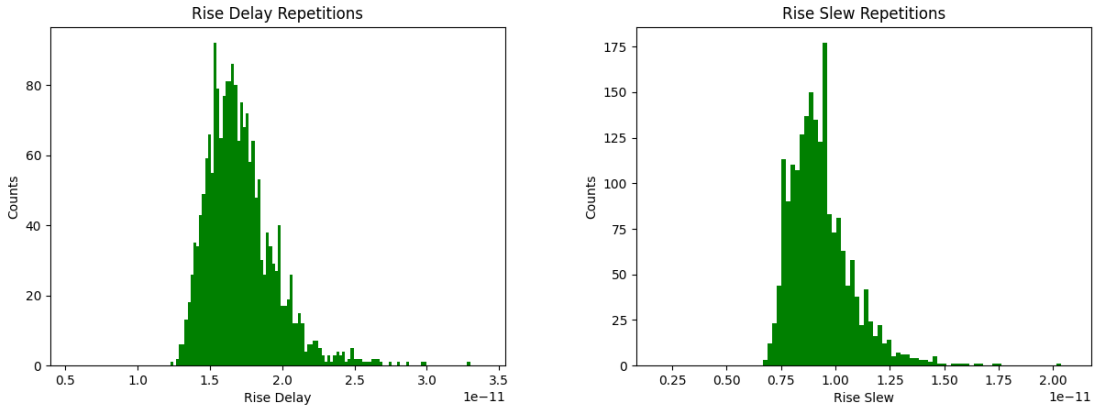
Like we mentioned above, for each different libcell, we present the experimental results in each subsection and cover both rise and fall input cases. After all the histograms are shown, there will be a comparison of the results within each subsection. Finally, at the end of this subsection, the differences between the three libcells tested will be explained.

For each subsection, buffer chains with 4 and 20 components are included.

5.1.1 BUFx2 Chain

4 BUFx2 - Rise Input

Figures 5.1a and 5.1b show the rise delay and slew of the last buffer in a 4 component buffer chain:



(a) 4 BUFx2 Rise Delay

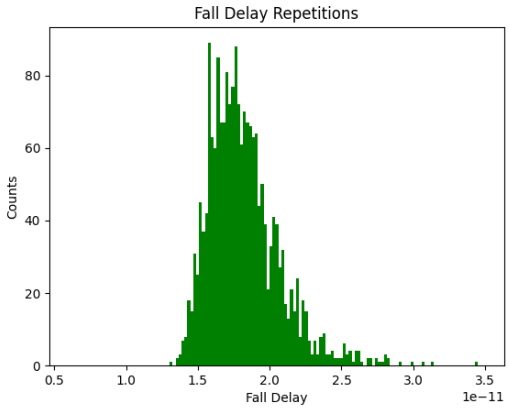
(b) 4 BUFx2 Rise Slew

Figure 5.1: 4 BUFx2 Rise Results

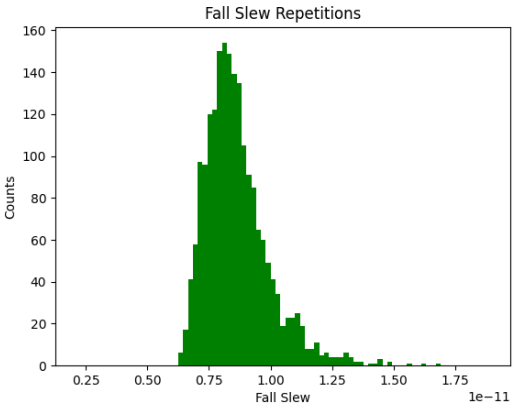
4 BUFx2 - Fall Input

Figures 5.2a and 5.2b show the fall delay and slew of the last buffer in a 4 component buffer chain:

From a first glance at the result histograms, the output delay measurements create a normal distribution (with the exclusion of the outliers) and the slew measurements from the multiple iterations create a slight lognormal distribution, which



(a) 4 BUFx2 Fall Delay



(b) 4 BUFx2 Fall Slew

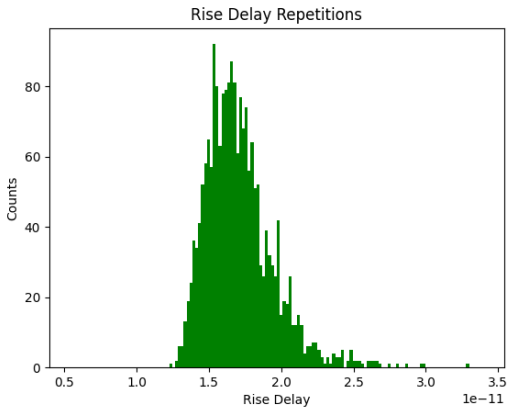
Figure 5.2: 4 BUFx2 Fall Results

is often expected, as noted by variation experts, since the input derate factors are generated randomly from a normal distribution. Also, it is worth noting that the fall delay is slightly higher than the rise one.

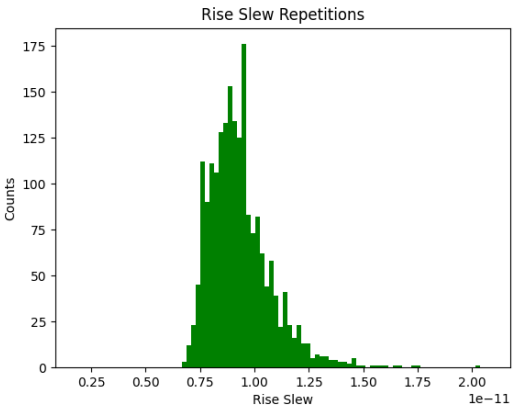
Note that these observations are present in all the following experiments, so they will not be mentioned again.

20 BUFx2 - Rise Input

We will now move on to the longer BUFx2 chains, containing a total of 20 components each. Figures 5.3a and 5.3b contain the rise delay and rise slew result histograms respectively:



(a) 20 BUFx2 Rise Delay

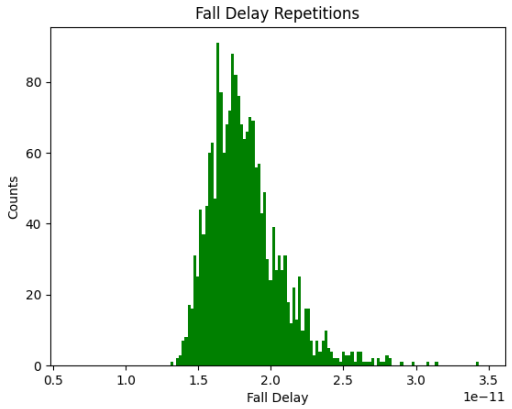


(b) 20 BUFx2 Rise Slew

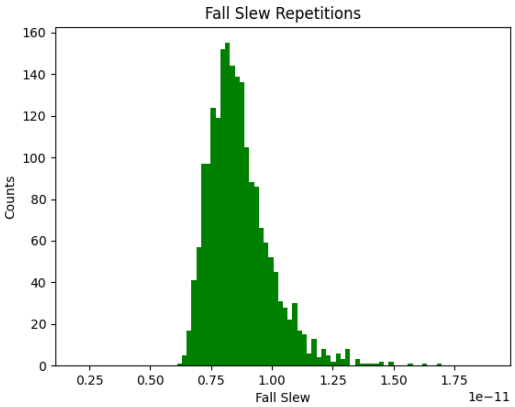
Figure 5.3: 20 BUFx2 Rise Results

20 BUFx2 - Fall Input

Similarly, we present the corresponding histograms for fall input (figures 5.4a and 5.4b):



(a) 20 BUFx2 Fall Delay



(b) 20 BUFx2 Fall Slew

Figure 5.4: 20 BUFx2 Fall Results

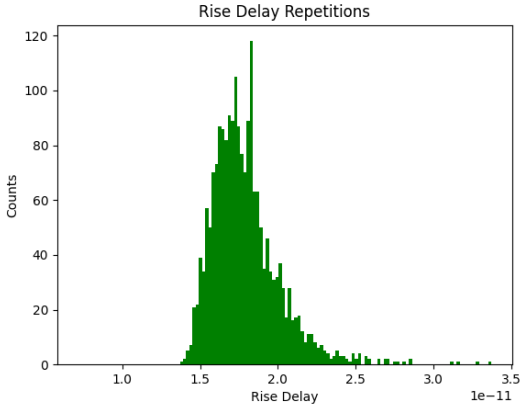
As we can observe, the results are almost identical to the smaller 4 BUFx2 chains. So, for the BUFx2 chains, we can assume that the number of components does **not** affect the overall distribution and behaviour of the results.

5.1.2 BUFx10 Chain

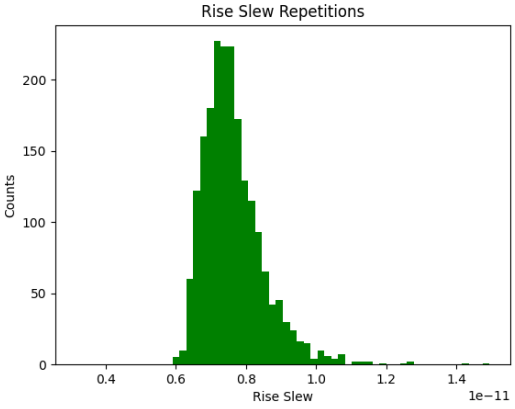
In this subsection, the same buffer chains will be examined, but with a different buffer libcell. Instead of BUFx2, a higher drive cell BUFx10 is used.

4 BUFx10 - Rise Input

Figures 5.5a and 5.5b contain the output result histograms for the BUFx10 rise input chains:



(a) 4 BUFx10 Rise Delay

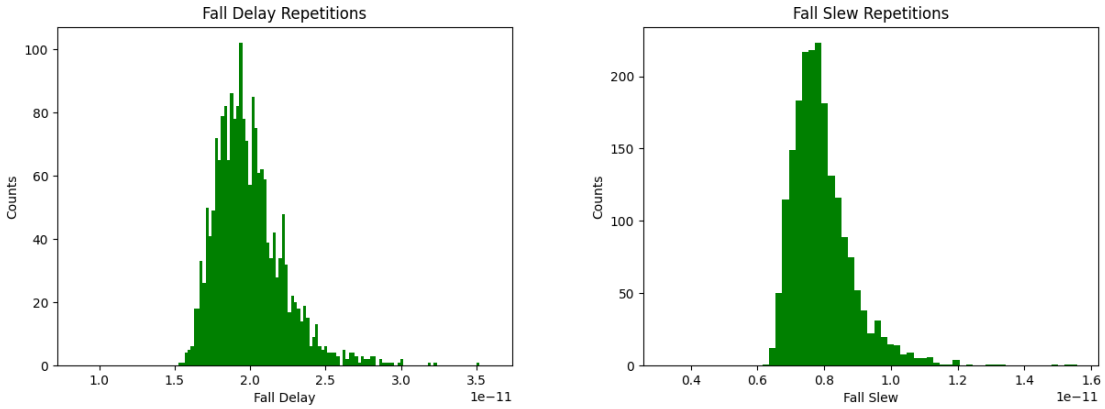


(b) 4 BUFx10 Rise Slew

Figure 5.5: 4 BUFx10 Rise Results

4 BUFx10 - Fall Input

Likewise for fall input (figures 5.6a and 5.6b):



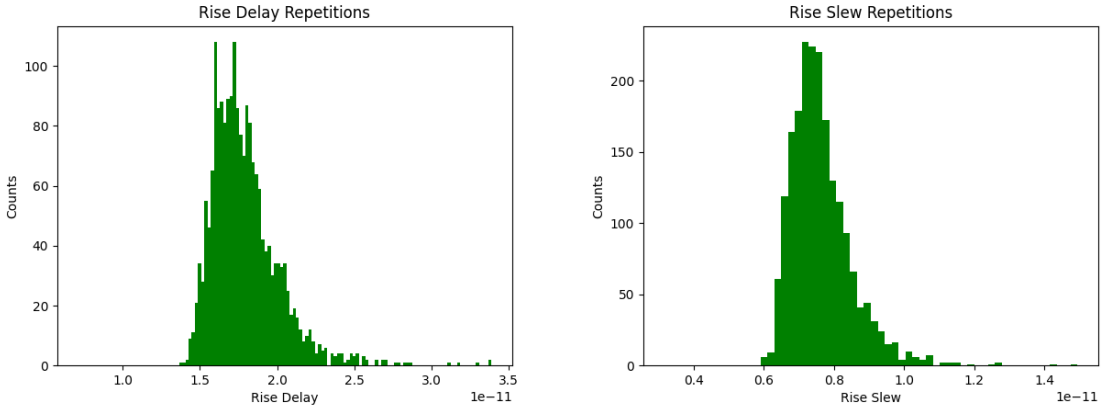
(a) 4 BUFx10 Fall Delay (b) 4 BUFx10 Fall Slew

Figure 5.6: 4 BUFx10 Fall Results

The same observations concerning the normal distribution of the delay results and the lognormal distribution of the slew results can be seen here as well, we can see that the mean and median values of each histogram in order, are higher than their respective ones in the BUFx2 chains. Again the fall delay is slightly higher on average, than the rise delay. However, while we expected both the delay and slew distributions to be a bit higher than the BUFx2 ones, due to the higher load, we can see that does not happen here. So, in this case, the difference in the load does not make much of an impact.

20 BUFx10 - Rise Input

In a similar manner, below we present the same results for the longer 20 BUFx10 chains. For rise input, the results are shown here in figures 5.7a and 5.7b respectively:

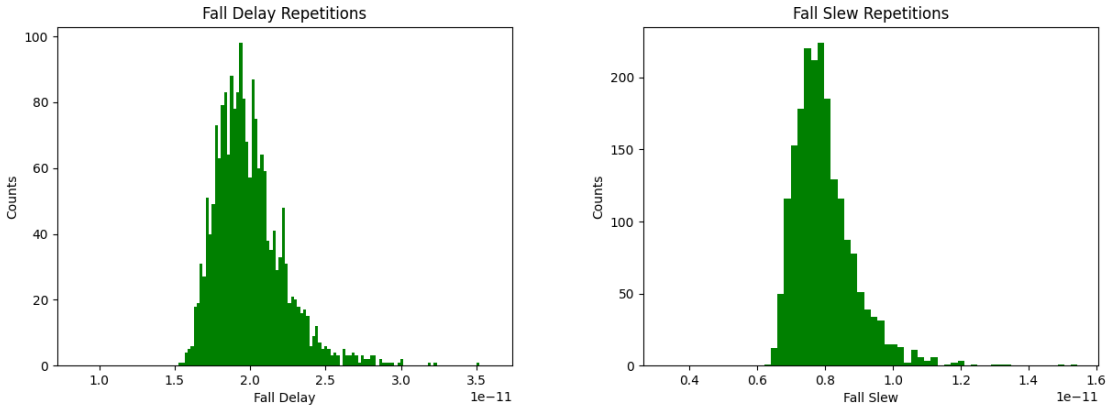


(a) 20 BUFx10 Rise Delay (b) 20 BUFx10 Rise Slew

Figure 5.7: 20 BUFx10 Rise Results

20 BUFx10 - Fall Input

Likewise for fall input (figures 5.8a and 5.8b):



(a) 20 BUFx10 Fall Delay

(b) 20 BUFx10 Fall Slew

Figure 5.8: 20 BUFx10 Fall Results

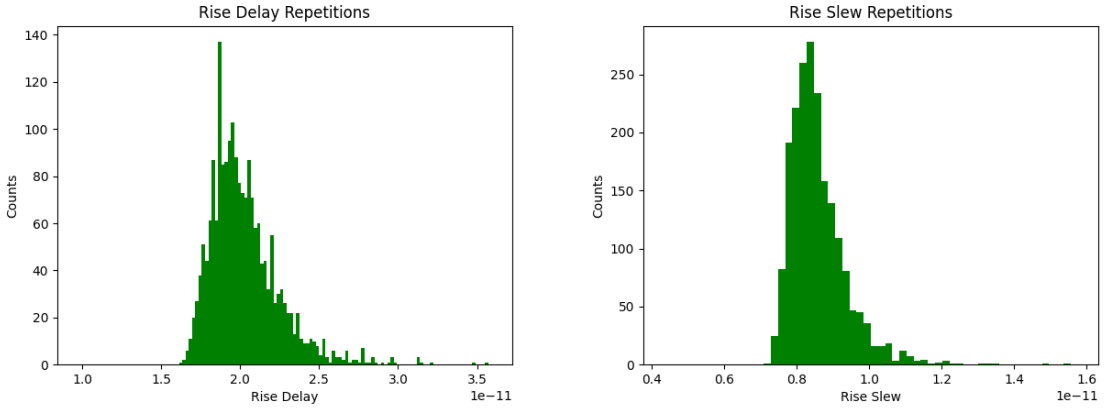
As we can determine from these results in figures 5.8a and 5.8b, the number of components in the chain does not alter the distribution of the results greatly, like we mentioned in the BUFx2 experiments. Other than that, the conclusions we came to in the smaller BUFx10 chains apply here as well.

5.1.3 BUFx24 Chain

Finally, we present the experimental results using the BUFx24 libcell.

4 BUFx24 - Rise Input

Figures 5.9a and 5.9b show the results for the 4 BUFx24 rise input chains:



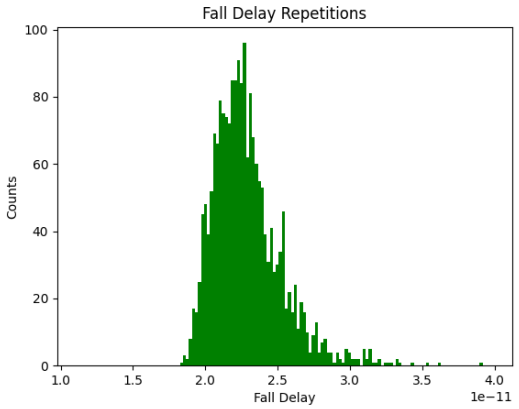
(a) 4 BUFx24 Rise Delay

(b) 4 BUFx24 Rise Slew

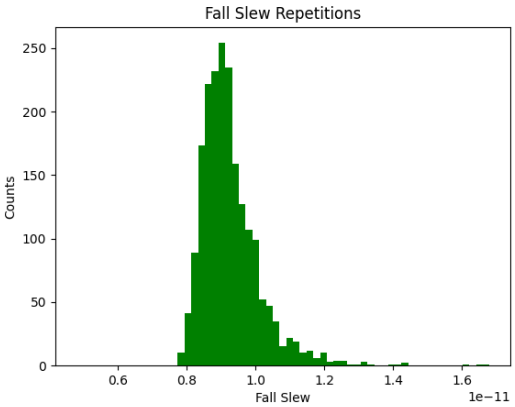
Figure 5.9: 4 BUFx24 Rise Results

4 BUFx24 - Fall Input

Likewise for fall input (figures 5.10a and 5.10b):



(a) 4 BUFx24 Fall Delay



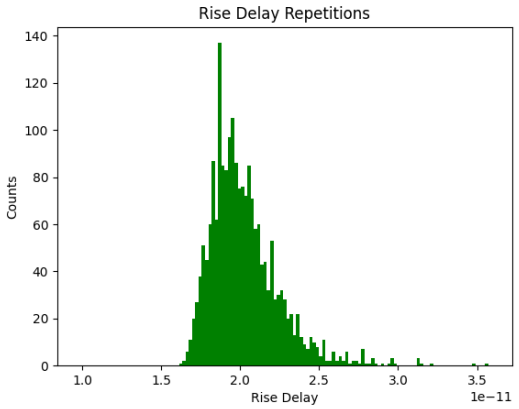
(b) 4 BUFx24 Fall Slew

Figure 5.10: 4 BUFx24 Fall Results

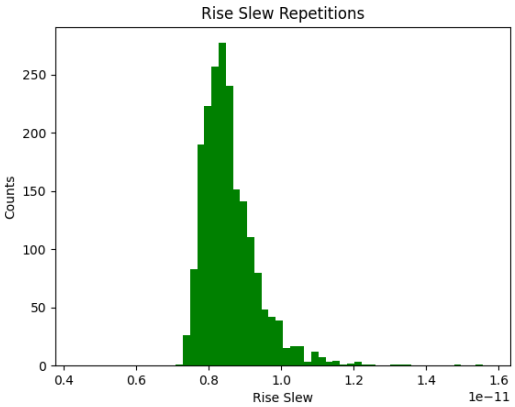
As expected due to the much higher load, the distributions are shifted to the right of the x-axis, meaning that both delay and slew values are higher. Now that we have tested all the three different libcells, it is worth mentioning that the higher the drive cell, the more lognormally skewed the distributions become, especially when it comes to the slew distributions. Now, let us examine whether in this final case for the buffer chains, the number of buffers affects the output delay and slew or not.

20 BUFx24 - Rise Input

The result histograms for the 20 BUFx24 rise input chains are shown in figures 5.11a and 5.11b:



(a) 20 BUFx24 Rise Delay

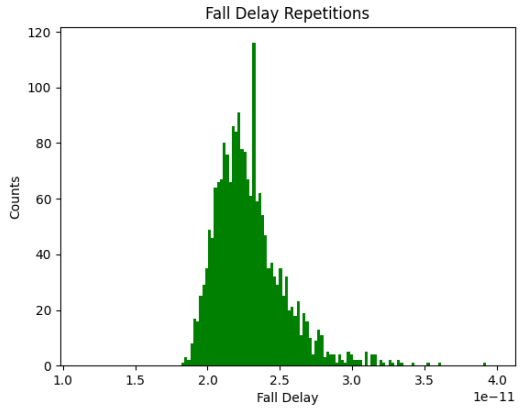


(b) 20 BUFx24 Rise Slew

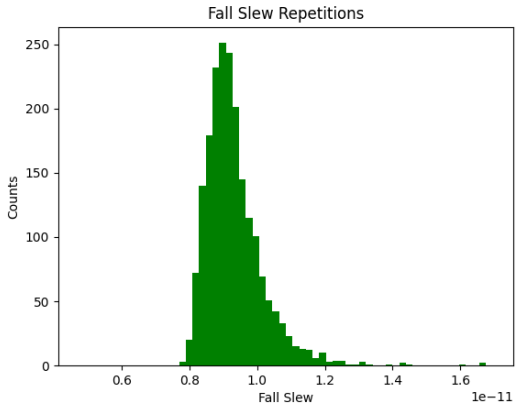
Figure 5.11: 20 BUFx24 Rise Results

20 BUFx24 - Fall Input

And finally, we present the 20 BUFx24 fall input chains (figures 5.12a and 5.12b):



(a) 20 BUFx24 Fall Delay



(b) 20 BUFx24 Fall Slew

Figure 5.12: 20 BUFx24 Fall Results

As we can observe by comparing the histograms in figures 5.11 and 5.4 with their respective ones in 5.9 and 5.10, we conclude that the number of buffers does not greatly affect the output results in any case.

5.1.4 Result Summary for Buffer Chains

From the above Monte-Carlo simulations, we examined the basic timing behaviour of different buffer chains. As such, we now have a general idea of the process variation of the delay and slew values by generating pseudo-random derate factors at the input of each circuit. Considering that the input derate factors are calculated from a **normal distribution**, as explained in chapter 4.1.2, from the experimental results presented, we come to the following conclusions about the timing of the circuits while simultaneously taking into account this simple process variation case created from the derate factors:

- The slew values measured at the output of the chain are distributed **lognormally** and **right-skewed**.
- Fall delay is slightly higher than rise delay.
- The number of components in the chain, **does not** greatly affect the output delay and slew values.
- The higher the buffer drive cell, the higher the delay and slew values and the more skewed the output distributions become.

These results can be further investigated, by testing more different components and circuits, in order for a designer to constrain the chip accordingly. In the next section, the experimental results of our second design type, the inverter chains, will be presented and analysed.

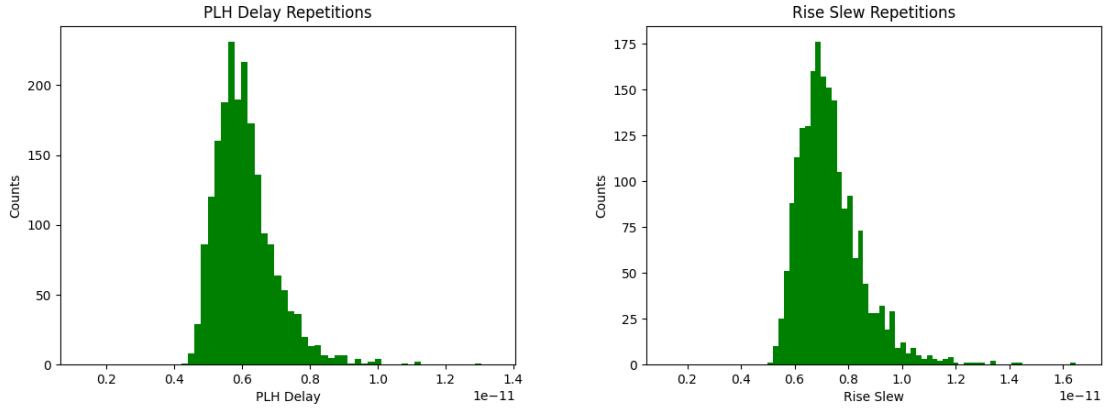
5.2 Inverter Chains

While the inverter chains are created from the same flow as the buffer chains, their output measurements are dependent on the number of components and the input signal. As such, we've also included chains with 21 inverters as well for the following experiments, to examine the case where the output signal is opposite of the input signal as well. Below we present all the results extracted from the inverter chains and analyse the output delay and slew values, as we explained them in 4.1.3.

5.2.1 INVx2 Chain

4 INVx2 - Rise Input

Figures 5.13a and 5.13b present the output distribution for the delay and slew values for the 4 INVx2 chains with rise input:



(a) 4 INVx2 PLH Delay

(b) 4 INVx2 Rise Slew

Figure 5.13: 4 INVx2 Rise Results

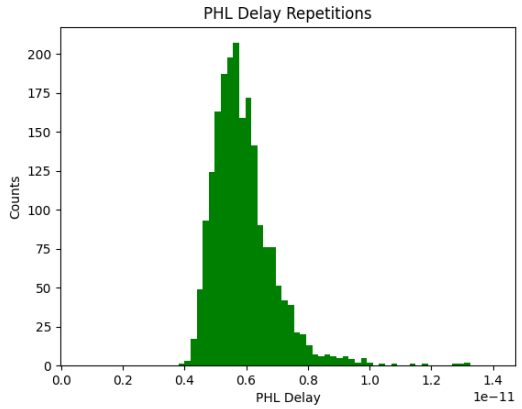
At first glance, we can observe that both graphs lean towards a slight lognormal distribution, while their average values are also lower than their respective buffer chains. It is worth mentioning that only for 4 INVx2 chains, the *SIGNIFICANTDIGITS* parameter for the results collection was reduced from 14 to 13, which is the same number set in the buffer chains. This was done in order to provide a more clear presentation of the results, due to higher variation.

4 INVx2 - Fall Input

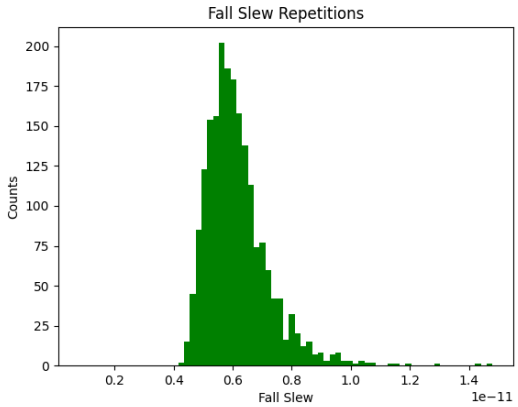
Below we present the same chains for fall input signal (figures 5.14a and 5.14b). While both distributions have a similar shape with the rise input ones, we can observe that their average delay and slew values are lower. This is especially visible in the slew graph.

20 INVx2 - Rise Input

Now let us try increasing the number of inverters to 20. In figures 5.15a and 5.15b the output delay and slew results of INVx2 chains consisting of 20 components



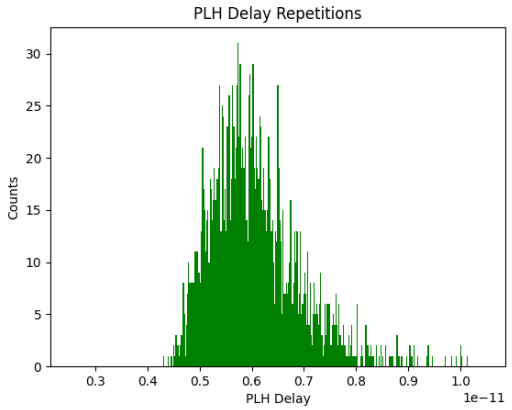
(a) 4 INVx2 PHL Delay



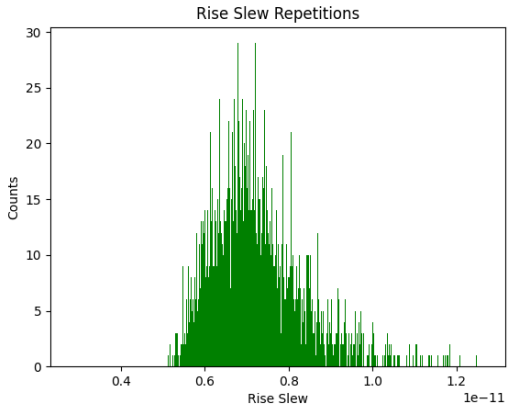
(b) 4 INVx2 Fall Slew

Figure 5.14: 4 INVx2 Fall Results

and rise input signal are presented. The *SIGNIFICANTDIGITS* parameter was set back to 14, since the observations are clearer now. Average values remain the same compared to the 4-component INVx2 rise chains. However, since we had to reduce the number of *SIGNIFICANTDIGITS* for the above case, we can assume that the variation here is slightly lower, albeit much higher than their respective buffer chains. The skewness of the distributions is also a bit lower, leaning towards a normal distribution.



(a) 20 INVx2 PLH Delay

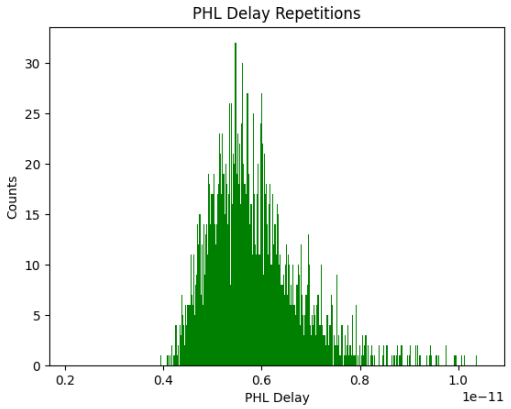


(b) 20 INVx2 Rise Slew

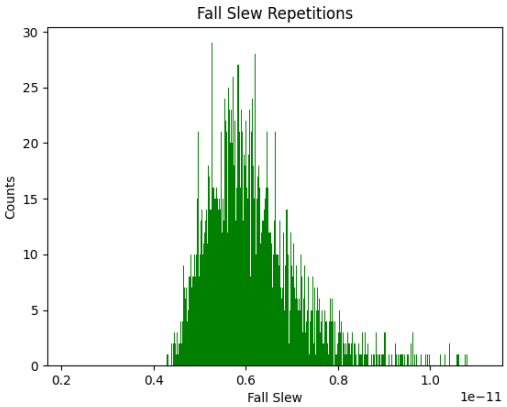
Figure 5.15: 20 INVx2 Rise Results

20 INVx2 - Fall Input

Let us present now the above values for the fall input 20-component INVx2 chains, as shown in figures 5.16a and 5.16b. As observed by the following graphs, the results are similar to the above cases: the average delay and slew values are lower than the rise chains -as seen in the 4-component chains as well- but the distributions lean more towards a normal distribution, similar to the 20-component rise chains.



(a) 20 INVx2 PHL Delay

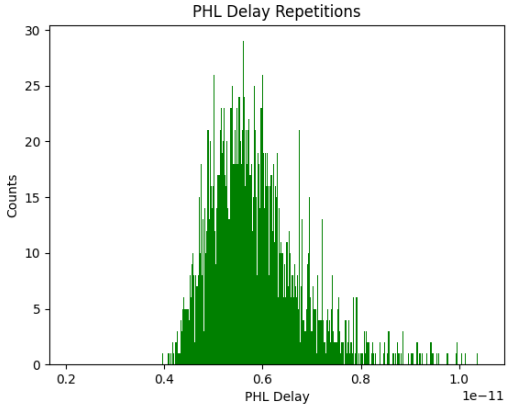


(b) 20 INVx2 Fall Slew

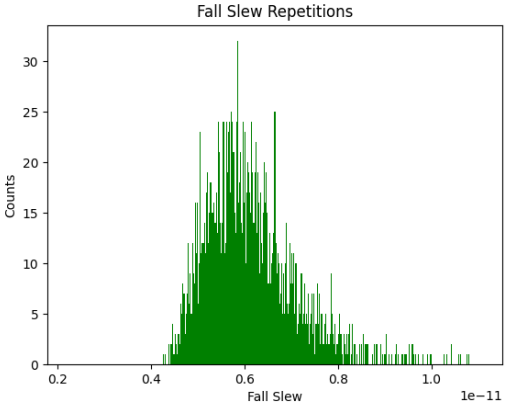
Figure 5.16: 20 INVx2 Fall Results

21 INVx2 - Rise Input

In figures 5.17a and 5.17b we present the output delay and slew values of the 21-component inverter chain with rise input. Note that the output values are the opposite of the previous rise input inverter chains, since there is one more inverter present.



(a) 21 INVx2 PHL Delay



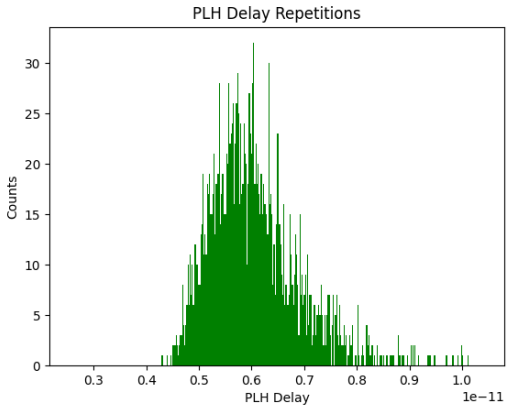
(b) 21 INVx2 Fall Slew

Figure 5.17: 21 INVx2 Rise Results

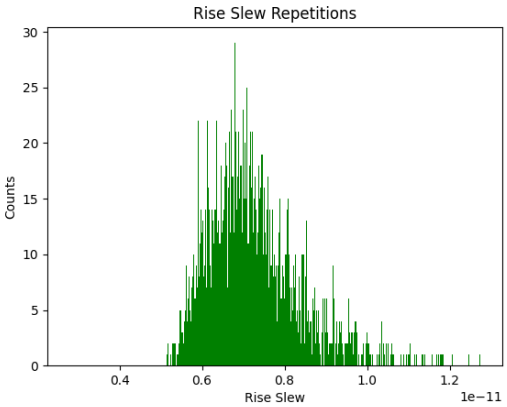
From the above graphs, we determine that the average delay and slew values are also dependent on the output signal, since the distribution are more similar to the previous fall input chains instead of the rise input ones.

21 INVx2 - Fall Input

Similar case with fall input, producing again the opposite values compared to the input (figures 5.18a and 5.18b. As observed in the exact above case, the results here follow the same pattern, as they are more similar to the rise input chains with even number of components.



(a) 21 INVx2 PLH Delay



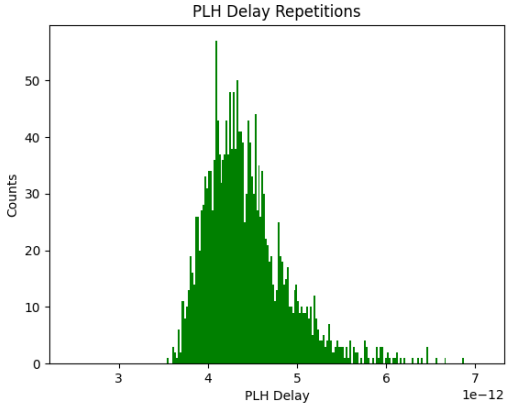
(b) 21 INVx2 Rise Slew

Figure 5.18: 21 INVx2 Fall Results

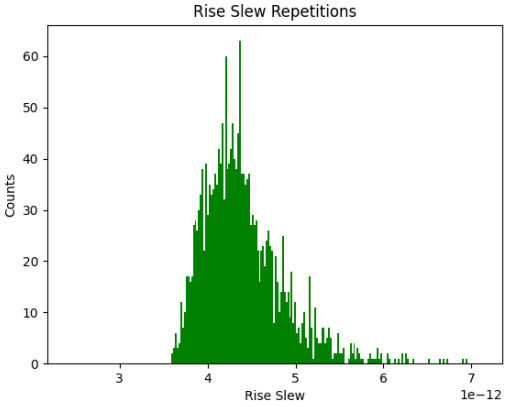
5.2.2 INVx8 Chain

4 INVx8 - Rise Input

The smallest of the INVx8 chains with rise input, produces these distributions as results (figures 5.19a and 5.19b):



(a) 4 INVx8 PLH Delay



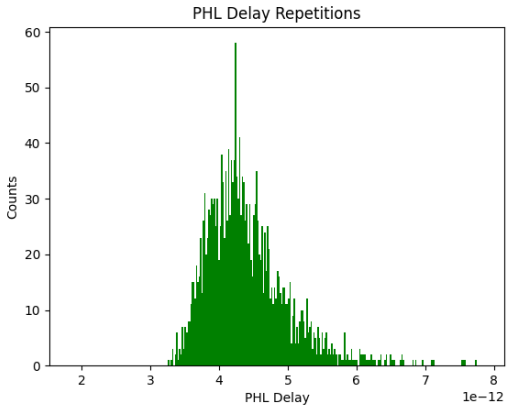
(b) 4 INVx8 Rise Slew

Figure 5.19: 4 INVx8 Rise Results

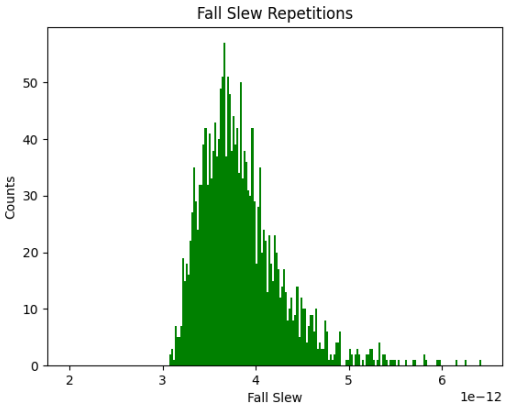
The distributions are slightly more skewed and lean more towards a lognormal distribution. However, both delay and slew values are relatively lower than the INVx2 chains, which is the opposite phenomenon we observed in the buffer chains, where the higher the input load, the higher the output delay and slew values as well.

4 INVx8 - Fall Input

In figures 5.20a and 5.20b we present the same chain with fall input. Similar to the INVx2 chains, average delay and slew values are lower than the ones in the rise input chains. The delay distribution is a little less skewed for this case.



(a) 4 INVx8 PHL Delay

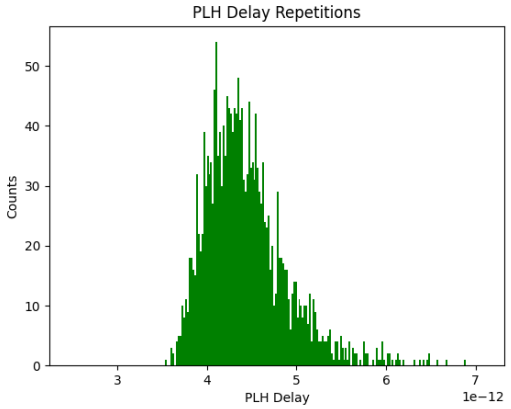


(b) 4 INVx8 Fall Slew

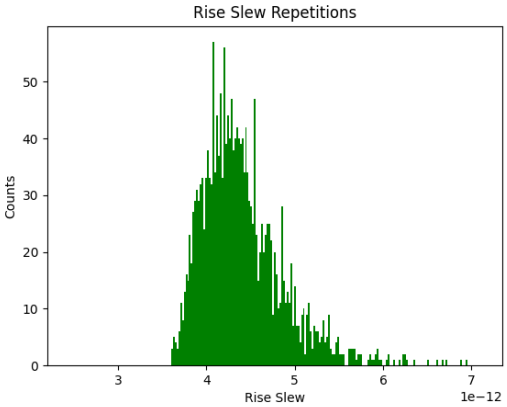
Figure 5.20: 4 INVx8 Fall Results

20 INVx8 - Rise Input

We will now present the larger INVx8 chains, starting from the 20-inverter rise input chain, as shown in figures 5.21a and 5.21b:



(a) 20 INVx8 PLH Delay



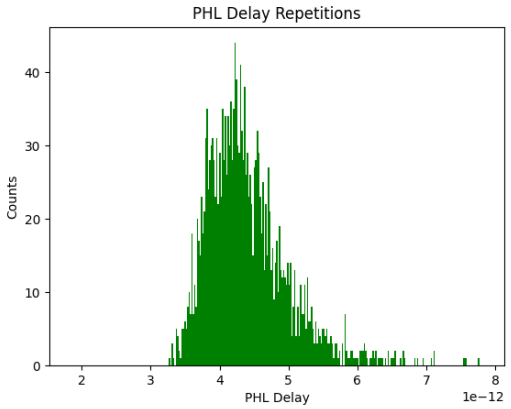
(b) 20 INVx8 Rise Slew

Figure 5.21: 20 INVx8 Rise Results

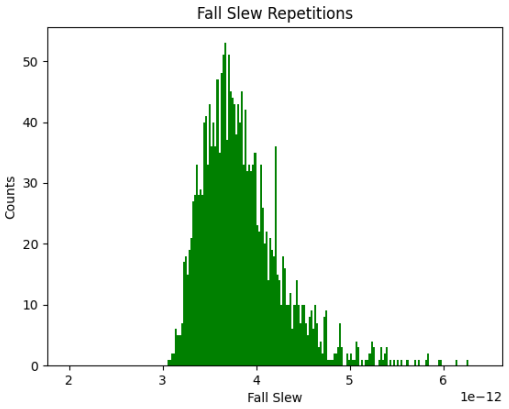
While in the INVx2 chains, we observed that the 20-component chains produced slightly less variation than the 4-component ones, that does not seem to be the case here, where the variation looks similar to the 4-component rise input INVx8 chains, with the average delay and slew values being almost the same as well.

20 INVx8 - Fall Input

In the equivalent fall input chains (figures 5.22a and 5.22b), we come at the same conclusions regarding the fall delay and slew values, which are relatively lower than the rise ones. Most of the distribution characteristics remain the same as the 4-component fall input inverter chain, with a slight exception in the delay distribution wich is a bit more skewed. So, the number of components does not affect the overall variation in this case either.



(a) 20 INVx8 PHL Delay

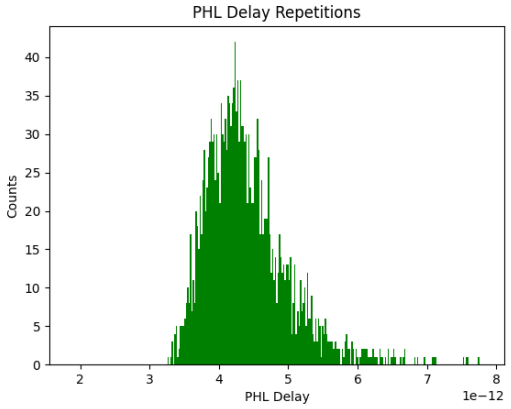


(b) 20 INVx8 Fall Slew

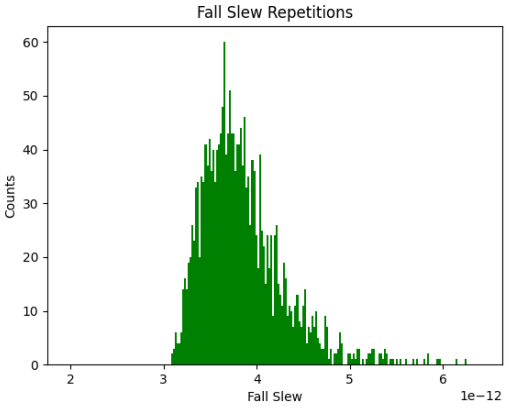
Figure 5.22: 20 INVx8 Fall Results

21 INVx8 - Rise Input

With the inclusion of an extra inverter in the rise input chain we have (figures 5.23a and 5.23b):



(a) 21 INVx8 PHL Delay



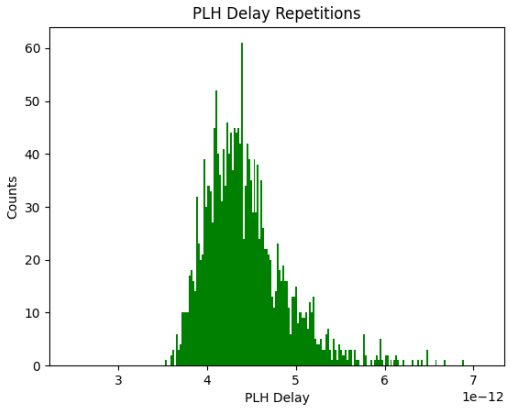
(b) 21 INVx8 Fall Slew

Figure 5.23: 21 INVx8 Rise Results

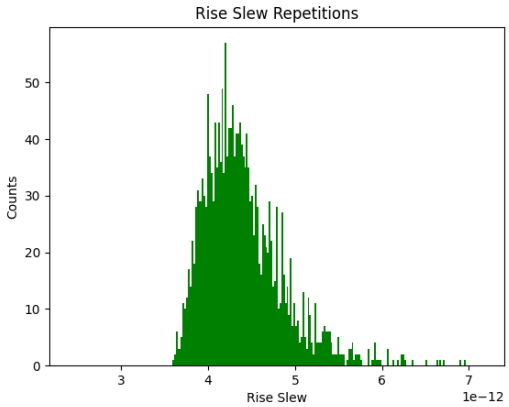
Similar to the 21-component rise input chain, the output values produced are the opposite ones (i.e. fall delay and slew) and the results are more similar to the fall even numbered INVx8 chains.

21 INVx8 - Fall Input

The same conclusion could also be drawn from the equivalent fall input chain, as seen in figures 5.24a and 5.24b:



(a) 21 INVx8 PLH Delay



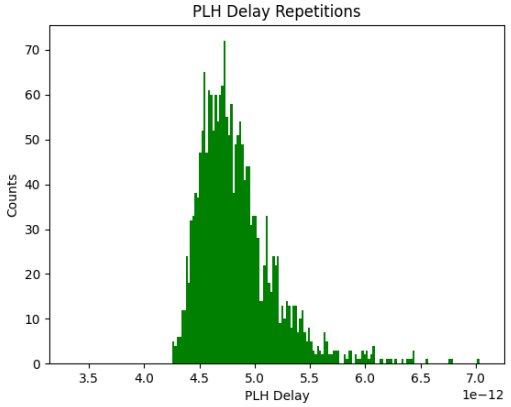
(b) 21 INVx8 Rise Slew

Figure 5.24: 21 INVx8 Fall Results

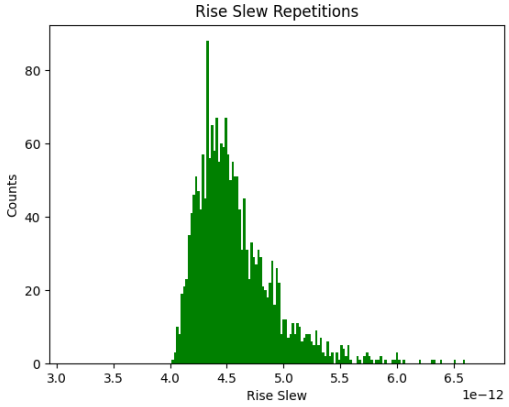
5.2.3 INVx13 Chain

4 INVx13 - Rise Input

We will now move on to the highest inverter libcells. Below we present the 4-component INVx13 rise input chain results (5.25a and 5.25b):



(a) 4 INVx13 PLH Delay



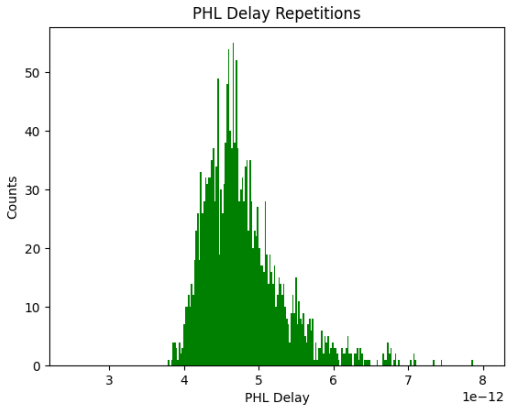
(b) 4 INVx13 Rise Slew

Figure 5.25: 4 INVx13 Rise Results

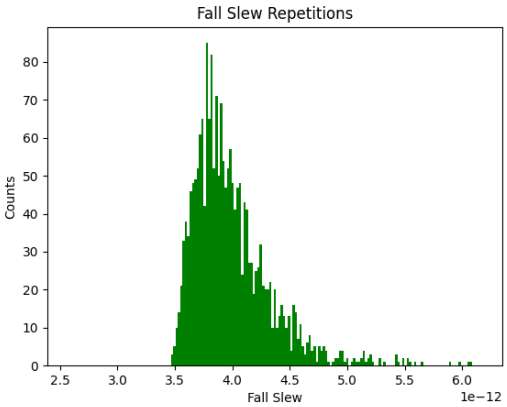
While the average values are closer to the INVx2 chains, the skewness of the distribution is higher, clearly leaninng towards a lognormal distribution.

4 INVx13 - Fall Input

The same findings can be observed in the fall input chain as well, i.e. distribution skewness is larger and the average delay and slew values are realtively lower than the rise ones (figures 5.26a and 5.26b).



(a) 4 INVx13 PHL Delay

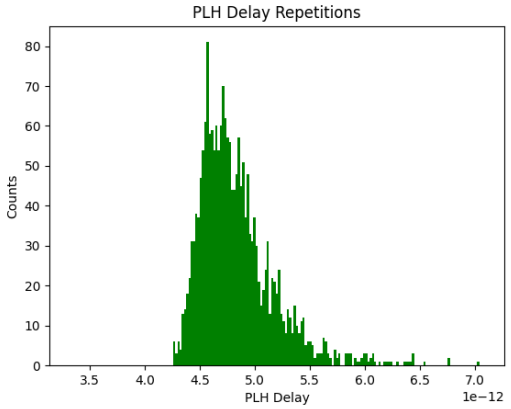


(b) 4 INVx13 Fall Slew

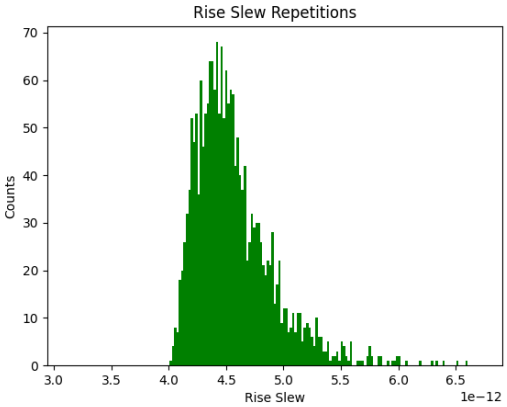
Figure 5.26: 4 INVx13 Fall Results

20 INVx13 - Rise Input

In figures 5.27a and 5.27b we present the results of the rise input INVx13 chains when increasing the number of components to 20:



(a) 20 INVx13 PLH Delay



(b) 20 INVx13 Rise Slew

Figure 5.27: 20 INVx13 Rise Results

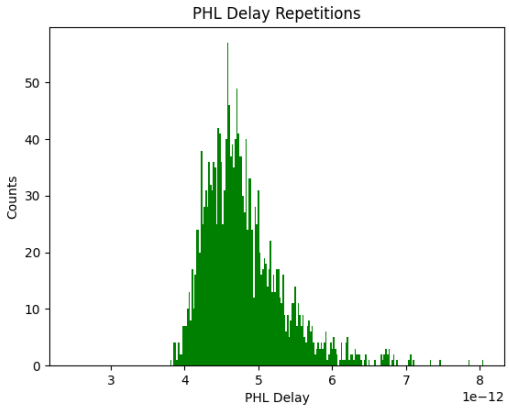
The distributions are almost identical to the 4-component rise input INVx13 chains, so the number of component does not greatly affect the variation.

20 INVx13 - Fall Input

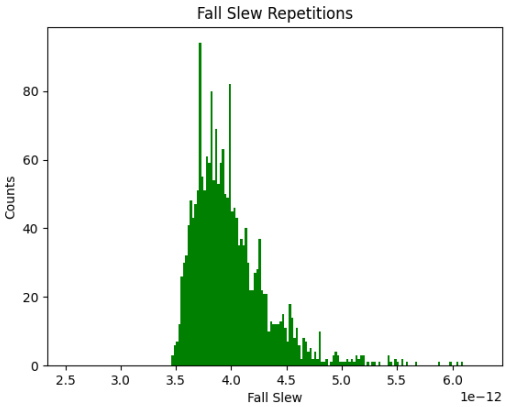
The same observations are present in the fall input 20-component INVx13 chains (figures 5.28a and 5.29b), which produce similar results to the 4-component equivalent chains.

21 INVx13 - Rise Input

By adding one more inverter and setting the input signal to rise, the following results are produced, as shown in figures 5.29a and 5.29b. The same conclusions



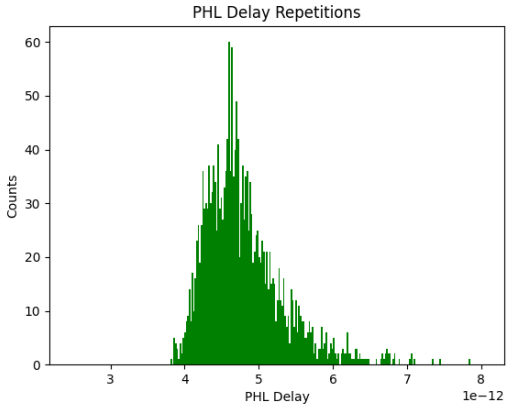
(a) 20 INVx13 PHL Delay



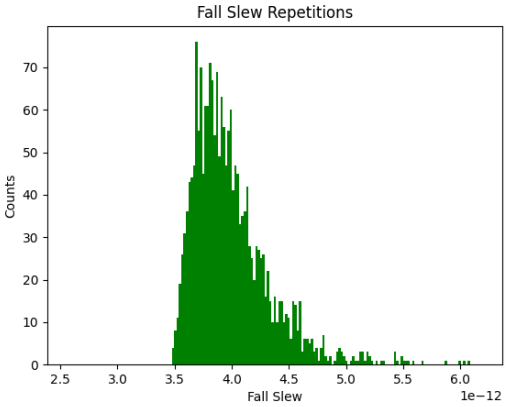
(b) 20 INVx13 Fall Slew

Figure 5.28: 20 INVx13 Fall Results

we reached by examining the previous 21-inverter chains, are present here as well, meaning that, not only the distributions are more skewed, but also the results are more dependent on the output signal.



(a) 21 INVx13 PHL Delay

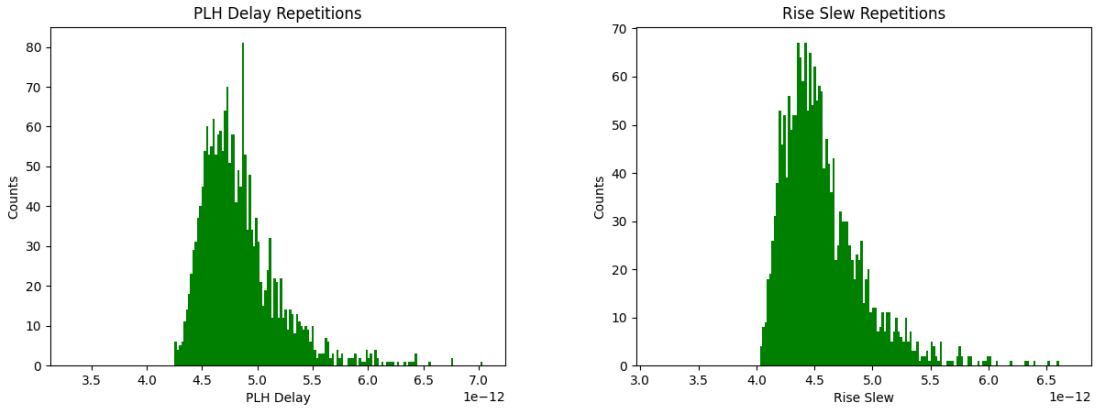


(b) 21 INVx13 Fall Slew

Figure 5.29: 21 INVx13 Rise Results

21 INVx13 - Fall Input

Finally, the 21-inverter fall input chain results (figures 5.30a and 5.30b), where we draw the same conclusions as before. Likewise, since the output signal is the opposite of the input, the results here are closer to the rise input even numbered inverter chains.



(a) 21 INVx13 PLH Delay

(b) 21 INVx13 Rise Slew

Figure 5.30: 21 INVx13 Fall Results

5.2.4 Result Summary for Inverter Chains

To summarise, by generating the same input derate factors, we performed the same simulation for different inverter chains and investigated their basic timing behaviour. As such, although there were some unique observations present in some cases, we have come to the following general conclusions for the inverter chain experiments:

- The output timing variations of higher libcells, while keeping a similar range of values, generate a more **lognormal** distribution than the lower ones.
- In most of the cases, similar to the buffer chains, the number of inverters present in the chain, **does not** greatly affect the output delay and slew values.
- The output values are also dependent on the type of the output signal, while fall (PHL) delays and slews are relatively lower than the rise ones. This could imply that a fall output of an inverter is slightly faster than a rise one.
- Inverter chains generally produced **more** variations than buffer chains.
- Higher inverter libcells produced **slightly less** variations than the lower ones.

Similar to the buffer chains, these results can be further investigated by testing more complex inverter circuits.

Chapter 6

Conclusion and Future Work

In this work, we created and presented an investigation-oriented transistor level Monte-Carlo flow to research process variation in simple digital circuits its effect on their timing analysis. By generating a distribution-based variation in the form of derate factors, we performed multiple simulations with this flow and presented the experimental delay and slew results in numerical distributions as well. While this flow currently only support simple basic circuits like the ones tested, it could be used as the basis to create more automated Monte-Carlo flows for more complex circuits, thus providing a more well-rounded approach in the field of SSTA. As such, this study could be expanded upon to integrate more and more statistical methods of timing analysis in the industry and thus, reducing the amount of pessimism of traditional STA and leading to the ovrall performance and functionality improvement of ASICs in the long run.

As part of our future work, the expansion of this flow to support circuits with a **fanout-of-4** is intended, as well as the implementation of a **Gate-Level Monte-Carlo** flow, which, instead of transistor level, applies derating in gate level of a specific component or library cell type. Gate-level Monte-Carlo, while less accurate, is much more computationally efficient, since the amount of SPICE simulations is greatly reduced. As such, a combinational method of transistor and gate level Monte-Carlo could be used, in order to easily apply derating and multiple simulations in much larger circuits, while also retaining the transistor level SPICE simulations for specified critical parts of the design, that require higher accuracy.

Bibliography

[1] "What Is Numerical Analysis?" [Online]. Available: https://www.mathworks.com/discovery/numerical-analysis.html

[2] "Direct and Iterative Methods." [Online]. Available: https://en.wikipedia.org/wiki/Numerical_analysis#Direct_and_iterative_methods

[3] "Monte Carlo Method Definition." [Online]. Available: https://en.wikipedia.org/wiki/Monte_Carlo_method

[4] "Probability Distribution." [Online]. Available: https://www.investopedia.com/terms/p/probabilitydistribution.asp

[5] "Probability Density Function (PDF)." [Online]. Available: https://www.investopedia.com/terms/p/pdf.asp

[6] "StatDist - Plot Distributions Online." [Online]. Available: https://statdist.com/

[7] "Normal Distribution." [Online]. Available: https://www.investopedia.com/terms/n/normaldistribution.asp

[8] "Mean, Median and Mode." [Online]. Available: https://medium.com/@nhan.tran/mean-median-an-mode-in-statistics-3359d3774b0b

[9] "Skewness." [Online]. Available: https://en.wikipedia.org/wiki/Skewness

[10] K.-H. Chang, *e-Design*. Boston: Academic Press, 2015. [Online]. Available: https://www.sciencedirect.com/science/article/pii/B9780123820389000107

[11] "Log-normal Distribution." [Online]. Available: https://en.wikipedia.org/wiki/Log-normal_distribution

[12] J. Bhasker and R. Chadha, *Static Timing Analysis for Nanometer Designs A Pratical Approach*. Springer, 2009.

[13] "Propagation Delay." [Online]. Available: https://chipedge.com/analysis-of-propagation-delay-in-vlsi-cmos-design/

[14] P. Das and S. K. Gupta, "Efficient post silicon validation via segmentation of the process variation envelope: Global vs. local variations."

[15] N. Blias, I. Lilitsis, S. Simoglou, E. Bakas, and C. Sotiriou, "Investigation on performance, power, area trade-offs using deterministic and monte-carlo process variation aware synthesis flows."

[16] M. Orshansky, J. C. Chen, and C. Hu, "A statistical performance simulation methodology for vlsi circuits."

[17] M. Merrett, P. Asenov, Y. Wang, M. Zwolinski, D. Reid, C. Millar, S. Roy, Z. Liu, S. Furber, and A. Asenov, "Modelling circuit performance variations due to statistical variability: Monte carlo static timing analysis."

[18] "Spectre Circuit Simulator User Guide."

[19] "The-OpenROAD-Project: ASAP7." [Online]. Available: https://github.com/The-OpenROAD-Project/asap7

[20] "SAMSUNG 14nm FinFET Process." [Online]. Available: https://semiconductor.samsung.com/us/support/tools-resources/dictionary/semiconductor-glossary-fin-field-effect-transistor-finfet-process/