ANALOG GENERATOR FOR A DRIVER OF A BOOST DC-DC CONVERTER

A THESIS SUBMITTED TO
THE FACULTY OF ARCHITECTURE AND ENGINEERING
OF
EPOKA UNIVERSITY

BY

ANDI GJINI

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR
THE DEGREE OF MASTER OF SCIENCE
IN
ELECTRONICS AND COMMUNICATION ENGINEERING

NOVEMBER, 2024

Approval sheet of the Thesis

This is to certify that we have read this thesis entitled **"Analog Generator for a driver of a boost DC-DC converter"** and that in our opinion it is fully adequate, in scope and quality, as a thesis for the degree of Master of Science.

_____

Assoc. Prof. Dr. Arban Uka
Head of Department
Date: November, 21, 2024

Examining Committee Members:

Prof. Dr. Gëzim Karapici (Computer Engineering) _____

Dr. Florenc Skuka (Computer Engineering) _____

Dr. Shkëlqim Hajrulla (Computer Engineering) _____

I hereby declare that all information in this document has been obtained and presented in accordance with academic rules and ethical conduct. I also declare that, as required by these rules and conduct, I have fully cited and referenced all material and results that are not original to this work.

Name Surname: Andi Gjini

Signature: _____

ABSTRACT

ANALOG GENERATOR FOR A DRIVER OF A BOOST DC-DC CONVERTER

Gjini, Andi
M.Sc., Department of Computer Engineering
Supervisor: Prof. Dr. Gëzim Karapici

This thesis explores the automation of analog circuit design, focusing on the driver of a boost DC-DC converter. It develops methodologies for automating schematic and layout design using tools like ANAGEN, WiCked, and Qgen. By transitioning from traditional manual methods to reusable, parameterized analog generators, the proposed approach significantly reduces design time and enhances scalability. The implementation demonstrates the effectiveness of automated optimization and layout strategies in meeting stringent design requirements, offering a practical solution for advancing analog design automation.

Keywords: *Anagen, WiCked, Qgen, Python, Schematic Generator, Layout Generator*

ABSTRAKT

GJENERATOR ANALOG PER NJE DREJTUES TE NJE KONVERTERI NXITES DC-DC

Gjini, Andi

Master Shkencor, Departamenti i Inxhinierisë Kompjuterike

Udhëheqësi: Prof. Dr. Gëzim Karapici

Kjo tezë eksploron automatizimin e dizajnit të qarkut analog, duke u fokusuar në drejtuesin e një konverteri nxitës DC-DC. Ai zhvillon metodologji për automatizimin e dizajnit skematik dhe të paraqitjes duke përdorur mjete si ANAGEN, WiCked dhe Qgen. Duke kaluar nga metodat tradicionale manuale te gjeneratorët analogë të ripërdorshëm, të parametrizuar, qasja e propozuar redukton ndjeshëm kohën e projektimit dhe rrit shkallëzueshmërinë. Zbatimi demonstron efektivitetin e optimizimit të automatizuar dhe strategjive të paraqitjes në përmbushjen e kërkesave të rrepta të projektimit, duke ofruar një zgjidhje praktike për avancimin e automatizimit të dizajnit analog.

Fjalët kyçe: *Anagen, WiCked, Python, Schematic Generator, Layout Generator*

ACKNOWLEDGEMENTS

Thanks to my professor, colleagues and whoever supported me.

TABLE OF CONTENTS

LIST OF TABLES

List of Figures

Listings

CHAPTER 1

INTRODUCTION

Nowadays the semiconductor industry has to respond to an ever-growing market for integrated circuits (ICs). The creation of increasingly complex chips, as predicted by Moore's law, and the increase in demand in every sector are the main challenges the microelectronics industry has to face. The exponential growth in chip complexity means a complicated and slow design process on the one hand and high costs and long time-to-market on the other.

The main objective for companies is to respond to the growth in demand of quality and efficient products, while keeping low time-to-market and costs.

If we take a detailed look at the chip manufacturing process, one factor that can cause delays is the presence of errors during the design phase, which then leads to delays that inevitably increase the cost of the design. In fact, finding of an error leads not only to the malfunctioning of the chip, but also to redesign and verify all the circuit characteristics, leading to a delayed production beginnings.

We can better understand the design cycle and the important steps that make it up, by looking at the pie chart below.

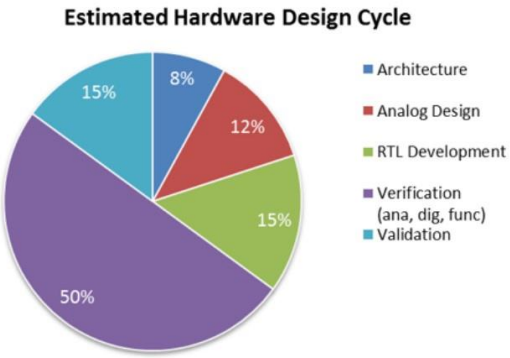

Figure 1: Estimated hardware design

As can be seen in Figure 1, the dominant part of the design cycle is verification, so if it is done carefully, costs and time can be minimised. In this phase all the necessary tests must be performed, involving the worst-case circuit working condition, varying several variables affecting the circuit, taking into account an ageing factor, etc., before the component is manufactured. To better understand why the verification is critical, it is necessary to know what the general workflow for the complete process is:

- Simulate the chosen design under typical conditions to ensure that all the specifications are met.

- Simulate again, but in corner conditions, i.e. varying temperature and voltage/current supplies between their minimum and maximum values, also taking into account process variation.

- Simulate again, but including the statistical distribution for process and devices parameters (mismatch).
- Draw the layout and get a netlist which includes parasitic resistances and capacitances introduced by the layout design.

- Perform point 1 to 3 using the extracted view got at point 4 instead of the schematic view.

In case of fails during one of these steps, a modification of the design is needed and all the verification cycle need to start from the beginning too. Only when all the tests are passed, through potentially a large number of iterations and loops, it is possible to move on to the fabrication phase and it is easy to see that the more complex the circuit, the more time this verification step requires. It is clear that by reducing the time needed for these expensive steps, the total cost of the design cycle will be reduced and a faster process will also get along with the always decreasing time-to-market request.

It must also be emphasised that both a digital and an analogue part can be found in a chip. As seen from the pie-chart, the analogue design is another important slice to consider in the process and also it represents the bottleneck of the design phase and contributes to the slowing down of the design process. Indeed, if for the digital part there are already advanced tools for design automation and verification, the same cannot be said for the analogue part.

The challenge is to develop something similar to what is already existing for the digital part, without forgetting the peculiarity and characteristics of the analog nature.

The use of specific computer-aided design (CAD) methodologies and tools are required to achieve an efficient result; in this sense, reuse-based design practices are regarded as a promising solution.

Therefore, in order to achieve low time-to-market and reduce costs for companies and at the same time automate the design process of analogue parts, like digital ones, automatic design tools and methodologies were adopted at Infineon, guaranteeing top-quality results.

This thesis proposes and develops an approach to schematic and layout design, through the use of framework and tools, such as Anagen, Wicked and H-Qgen. The aim is to move from a traditional method, in which the focus was on the design of individual instances that could not be reused and were created specifically to meet certain specifications, to a design that focuses on the creation of instance generators, called analog generators.

Thanks to these generators, starting from a specific topology it is possible to obtain IP blocks dimensioned according to the specifications provided as input. In the case that the design specifications have to be modified, the re-design takes place very quickly when compared with the time required by the traditional methodology.

It can therefore be stated that analog generators captures the designer methodology and aim to speed up the design process and at the same time to be an assistance tool, capable of providing blocks reliably sized according to the specifications.

Generators must be designed for both schematic and layout, starting with the chosen topology. The following paragraphs present the tools that have enabled the realisation of such a generator.

1.1 ANAGEN

ANAGEN is a hierarchical design framework written in Python: the aim is to automate the design of analogue blocks in analog mixed circuits through the realisation of generators that are independent of process and technology [1].

Started with open source Python framework BAG (Barkley Analog Generator) which is a modular collection of scripts helping in design automation from Berkeley Wireless Research Center, Infineon has decided to develop its own specific flow: ANAGEN. It is more than a framework, it is a collection of tools/APIs to do design automation in an Infineon environment whose objective is to enable users to create, use and test process portable analog generators. The ANAGEN program enables a combined usage of algorithmic design optimization (gm/ID, Wicked), different layout generator engines (Berkeley xBase, Infineon Qgen) together with traditional design, verification flows and tools within the Infineon Camino design flow.

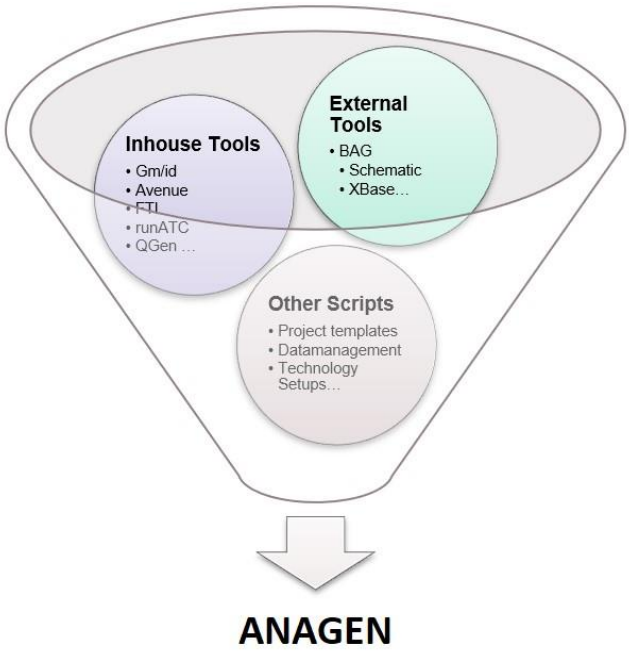

Figure 2: ANAGEN

As can be seen in the Figure 2, the subflow aims at bundling all tools necessary for generator development. The designer can create a template of a schematic and write a script which encapsulates the sizing process for the devices of the circuit, also known as the methodology, in a parametrized way. This creates a win-win situation in that designers and layouters can focus on essential tasks and spend less time on standard tasks, even if custom analog designers have been very resistant to this style of automation. The reasons are different, starting with the resistance to learning coding to the difficulty of analogue design as it changes in each circuit. The main difficulties encountered with this change in design methodology with ANAGEN are as follows:

- The complexity of geometric constraints: Analog layouts require precise matching and symmetry, and layout engineers use clever techniques based on human intuition and expertise. To match this, algorithmic methods must cover a large search space, and this has been infeasible in the past
- The absence of unifying performance metrics: unlike digital designs, which are characterized by power, performance, and area (PPA), the performance specifications of analog circuits are different for each class of circuits
- The wide variety of circuit classes and topologies: there is no equivalent for the relatively compact standard cell libraries used for digital design, and even basic analog building blocks can be constructed in a large variety of ways.

As mentioned earlier, ANAGEN follows a programmatic approach, i.e. it does not deliver instances, but rather captures designer’s best approaches in generators, in such a way as to overcome all the difficulties exposed.

The generators focus on schematic, layout and verification, implementing the concept of reusability combined with automation. Productivity gains come from parametrization, incremental extension and process portability. With all these features, the impact of ANAGEN in the design process is more than positive, speeding up key steps.

Through the creation of a reference library of designs, with a simple selection of the concept concerned and readjustment to the new work, a great deal of time is saved, as well as costs; however, everything must be well organised and well documented.

As it is easy to understand, the design methodology allows for the preservation of the modus operandi, rather than a specific product.

Finally, optimisation algorithms, obtained through a mix of the field of expertise and knowledge-based optimisation methods, can solve complex but definite problems. ANAGEN is written in Python using the object-oriented programming (OOP) model, organizing the software design around data or objects, instead of functions and logic. This approach also reflects the concept of reuse, since it embeds a hierarchical structure, making the reuse of the code itself easier through objects with relationship like parent-child.

ANAGEN provides in details:

- ANAGEN server, which contains already created generators that can be installed from Cadence-Virtuoso directly.
- ANAGEN technology setups, which allow the definition of a device flavor[1]
- Python API[2], which is very powerful and user-friendly, and also provides lots of functional packages and classes for the generators
- Python Wrapper, which makes it possible to run simulations in Cadence-Virtuoso
- Example Generators: There are many open-source templates and examples

The framework structure can be seen in the Figure 3.

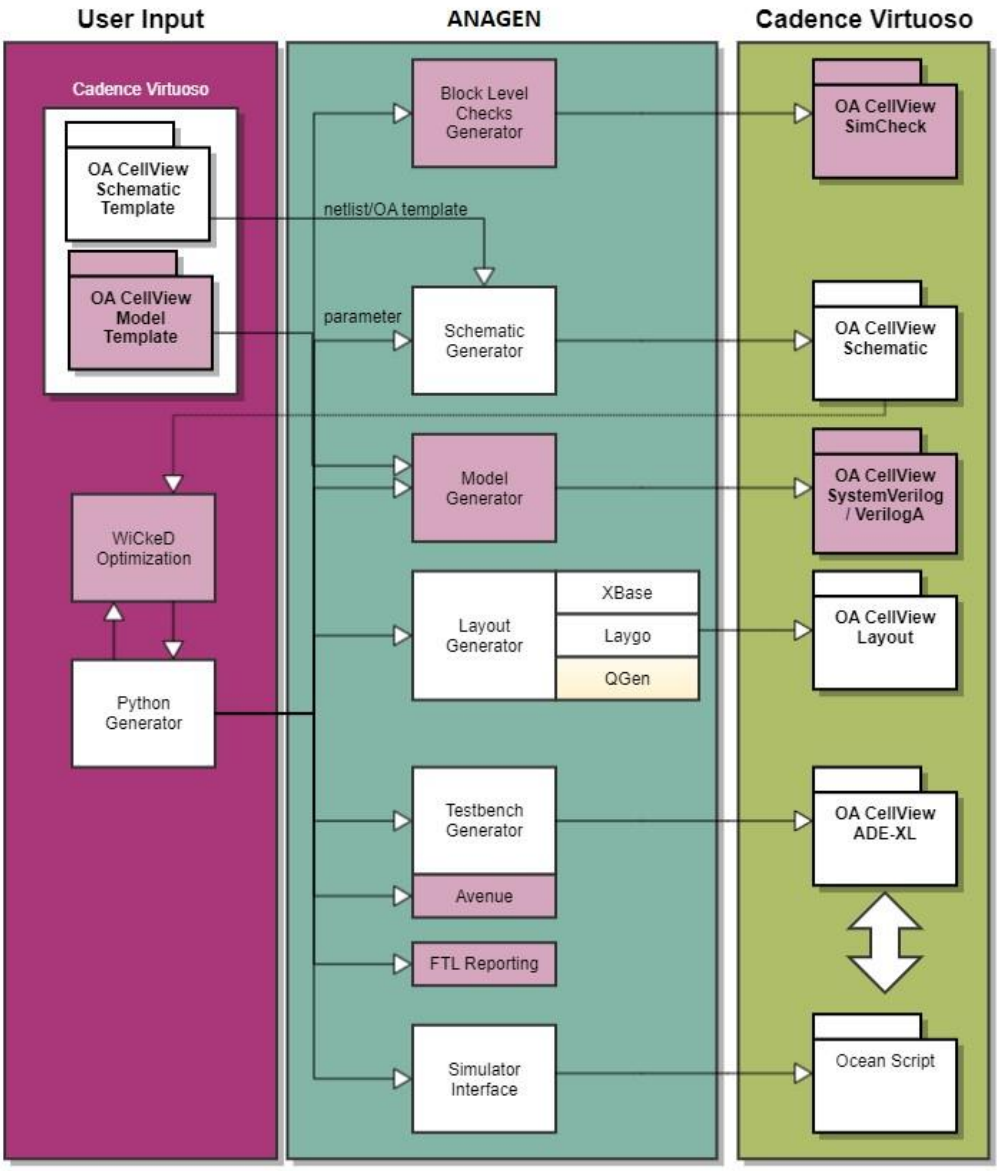

Figure 3: Anagen Framework

The starting point to create a generator is the topology of the circuit, which will not be alterated or manipulated, but it remains fixed.

As already mentioned, this framework provides the implementation of various functions such as dimensioning, schematic implementation, layout and verification: as can be seen from Figure 1.3, the Python generator, which is the actual user input, provides the various instance generators together with the verifications, which in turn communicate with the Cadence design environment, i.e. Virtuoso. Generator are written in Python and these scripts communicate directly with the users Virtuoso instance, thanks to low-level ANAGEN libraries via SKILL commands, while the generated instances are available immediately in Cadence-Virtuoso for debugging and tuning.

In the ANAGEN version, the blocks highlighted in pink in Figure1.3 constitute the upgrades made from the previous version. Of particular interest is the Wicked optimization block, which makes it possible to optimize the schematic implemented by the generator (more details in the next paragraph).

A simplified version of the framework structure is presented in Figure 4.

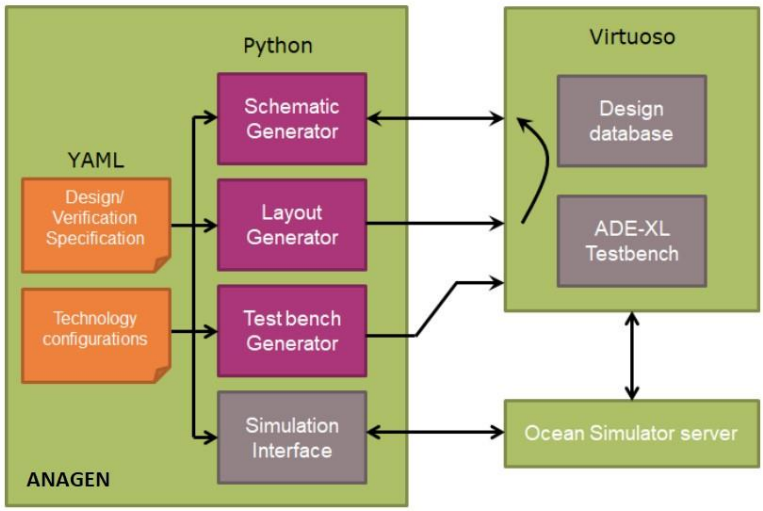

Figure 4: ANAGEN Framework simplified

As can be seen in this simplified version of anagen illustrated in Figure1.4, the main blocks of the framework are those who permit the coding of generators, specifically: schematic generator, layout generator and testbench generator.

Therefore ANAGEN provides all the tools needed to develop a generator of a circuit topology specified as input. However, attention must be paid to the flow to be followed to create a generator. The complete flow is presented in Figure 5.

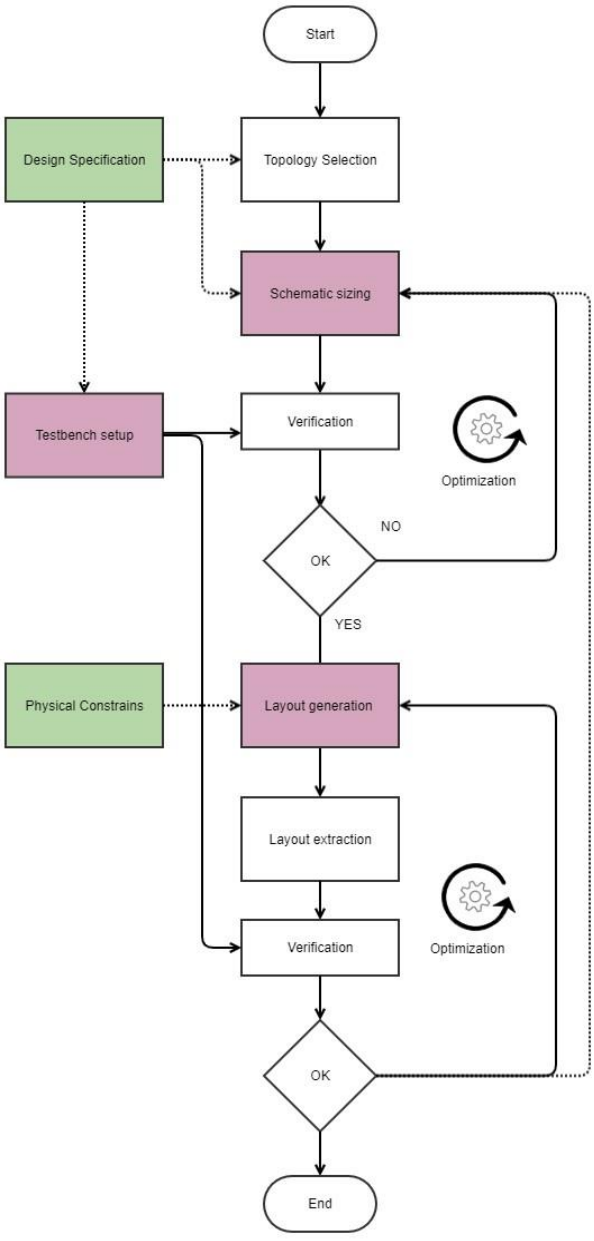

Figure 5: Anagen Workflow

It is important to visualize the flow that has to be followed in order to produce a reliable generator, while noticing that verification and optimization can also be included in the schematic generator itself. If something is not good enough at the schematic verification then it is not possible to proceed with the layout and an optimization step is needed to achieve a performing circuit.

The flow embeds the design approach to be adopted at programming level, which consists on four important parts. The initial setup phase in which:

- Create a template of the intended circuit
- Parametrize properties (transistor length and width, capacitors and resistors size)
- Create a verification setup

Then it proceeds with the second phase of initial sizing:

- Translate the system requirements into circuit requirements
- Perform initial sizing with the gm/ID methodology or other suited methogology, implementing the schematic generator

After that, the third phase regards the optimiser:

- Define optimization objectives and constraints
- Run the optimizer on the constrained sizing problem
- Typical run times are in the order of minutes / few hours

After the phase concerning the design and optimisation of the schematic is finished, the last phase proceeds with the layout:

- Execute a layout generator
- Run design rule check and perform parasitic extraction

- Run final verification, and IP release

If problems are found in the verification phase, the layout phase is repeated, otherwise the optimisation with the initially identified constraints is carried out also for the final part.

So as one can guess this process is iterative, each time a specification is not met, the flow has to go back to the schematic or layout generator and change some parameters according to the script, as a designer would do by hand. Once the topology has been selected and the relative schematic has been produced with devices belonging to ANAGEN, therefore of possible manipulation through Python code, it is possible to start a project with a Python template inside the integrated development environment IDE. This file.py contains the class generator in which all the coding strategy for the sizing of the schematic is written. Instead, for the coding of the layout generator, it is necessary to refer to other javascript language files. For this thesis only the schematic sizing has involved the use of ANAGEN, while the tool that has been used for the layout is Qgen, which currently is independent of ANAGEN. So to summarise, the main objective remains the development of generators that deliver quality results and reduce time-to-market. This can partially be achieved through the coding of a methodology based on the experience and knowledge of the designer, and also through optimization algorithms, either coded at the moment or imported from existing optimization tools (e.g. MunEDA WiCked, pymoo etc.).

1.2 WiCked

WiCked is a tool that is part of a larger group of EDA solutions provided by the software company MunEDA [2]. This is a software suite that adds more functionalities to existing EDA design environments. The acronym EDA stands for *Electronic Design Automation* and can be defined as the category of tools used for designing and manufacturing electronic systems, from printed circuit boards to integrated circuits. The tool can be used for different purposes, for instance the analysis and optimisation of the yield and performance of analogue, mixed-signal and digital designs [3].

MunEDAs tools provide different skills for IP porting, re-sizing, re-targeting, analysis and verification, modelling and even sizing and optimization for full-custom design in advanced technologies.

WiCkeD helps the circuit designer to improve the efficiency, decrease the risk of failures and increase the circuit quality, through the consideration of statistical variation effects of modern process technologies.

Several tools and features of this environment are made available.

Here is a list of them:

- FEO - *Feasibility Analysis and Optimization* - Defines and analyses the circuit's functionality based on the electrical, layout, area and others constraints running automatically netlist parametrization and optimizes and fulfills them automatically for best functionality

- DNO - *Deterministic Nominal Optimization* - Sensitivity based circuit optimization for nominal (typical) case and worst case operating corners

- GNO - *Global Nominal Optimization* - Statistical and stochastic circuit optimization based on sampling and design-space exploration search algorithms

- YOP - *Yield Optimization* - Automated circuit yield and robustness optimization for high sigma and performance margins
- REL - *Reliability Option* - includes and considers available reliability models and constraints for enhanced aging, degradation, area and other reliability influence factors

The tool can be configured and executed in script-based (batch mode) or fully GUI-driven. Optimization is based on complete specifications and constraints are managed by the tool itself, trying to satisfy all of them. It is proven in wide range of technologies from 350nm CMOS down to 5nm FinFET advanced node PDK.

We can make a list of benefits which ranges from automatic performance tuning for all types of analogue and mixed-signal circuits (amplifiers, transceivers, PLLs, oscillators, mixers, data converters, filters and many more) to optimisation of performance, robustness and high-sigma.

Consequently we can mark improvements in different fields such as performance, power, area, delay, as well as parasitic effects and design time effort.

Since March 2021, WiCked has been used productively as a mathematical plug-in of Infineon IP Factory to statistically characterise standard cell. The challenge is to achieve a high accuracy by using various machine learning techniques, such as worst-case analysis, interpolation, adaptive determination of interpolation points and many more. In order to reduce the overall execution time, parallel processing and simulations are used.

Figure 6 shows the tasks that the optimizer is able to provide; for this thesis, circuit optimisation at PVT corners is of interest.

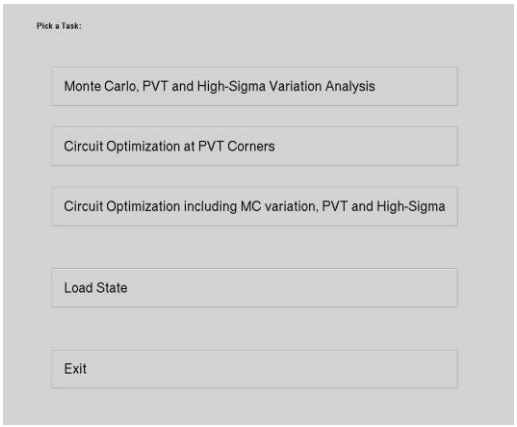

Figure 6: WiCked Tasks

After selecting the task of interest, if the session is started in batch mode, the netlist for the project of interest is selected manually, otherwise it will be selected automatically. In the constraint editor window, one can set the optimiser according to interest case and parameterise the parameters of interest and set acceptable ranges for them, set pvt corners (process, voltage, temperature) combining them with acceptable temperature and power ranges. Finally, in the section performance there are all the measurements to be made with the type of analysis (dc, tran...) as well as the possibility of adding area measurement.

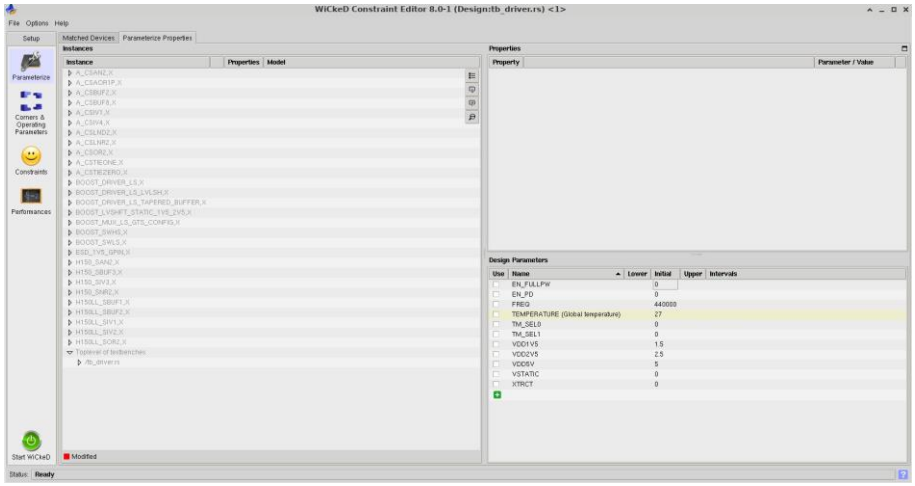

Figure 7: Constraint Editor

After setting up the optimiser, it can be launched. This opens the WiCked GUI as in Figure 8, where the tools of interest can be used. For the thesis, three tools are used:

1. Simulation: in which a nominal simulation of the circuit is made and it is checked whether the results obtained are those expected

2. Worst-case Operation & Corner: the circuit is simulated in such a way as to identify which combination of corners (temperature, power supply, process...) leads the circuit to the worst behaviour, in this way it can be seen how performance degrades as the conditions of interest change

3. DNO - Deterministic Nominal Optimisation - the actual optimisation is carried out, after setting performance bounds or specifying a minimisation; the tool varies the parameters that were set in the previous phase and provides suitable values to be inserted into the circuit for it to behave in the optimum way.

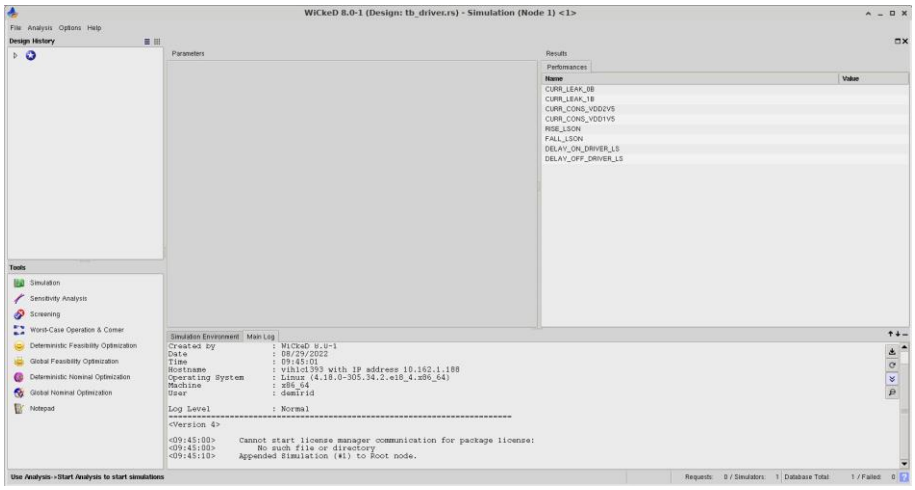

Figure 8: WiCked GUI

For this thesis the optimiser has been used, but not included in the ’automatic’ flow as this phase is still in the development phase. The results obtained using WiCked will however be presented in the following chapters.

1.3 Qgen

Infineon Technologies has developed an in-house layout module generator, called Qgen. This tool is written in C++ using the Qt[1] framework, whereas technologies setups and generators, called from C++ interface, are written in JavaScript.

A first problem regards the integration of the use of JavaScript in the flow, since the ANAGEN framework is written in Python, and therefore schematic generator and the layout generator cannot be put in a complete automatic design loop.

Qgen is the chosen tool for this thesis work because it is able to deal with the used technology and also for different reasons [4]:

- Reduce layout implementation time compared to the one made by hand.
- Fast exploration of instance and layout parameters.
- Qgen instances are OA figGroups (instances of PDK PCells[2], Vias, shapes)
- Formally correct (e.g. no undesired shorts created).
- Layout XL compliant.

[1] Qt is a widget toolkit for creating graphical user interfaces as well as cross-platform applications

[2] A PCell is a parametrized cell which represents a part or a component of the circuit whose structure is dependent on one or more parameters

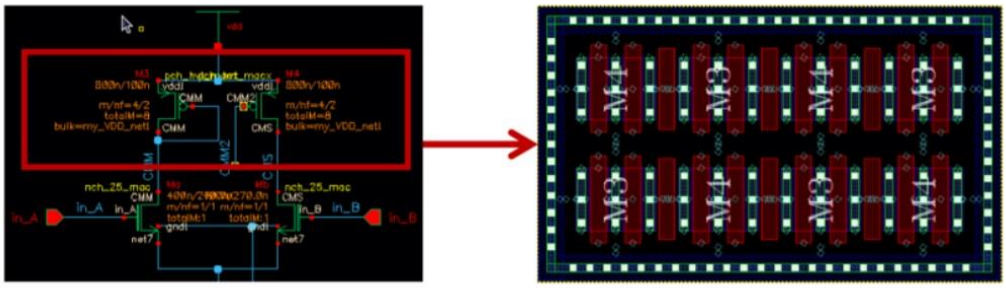

Figure 9: Qgen transistor matrix

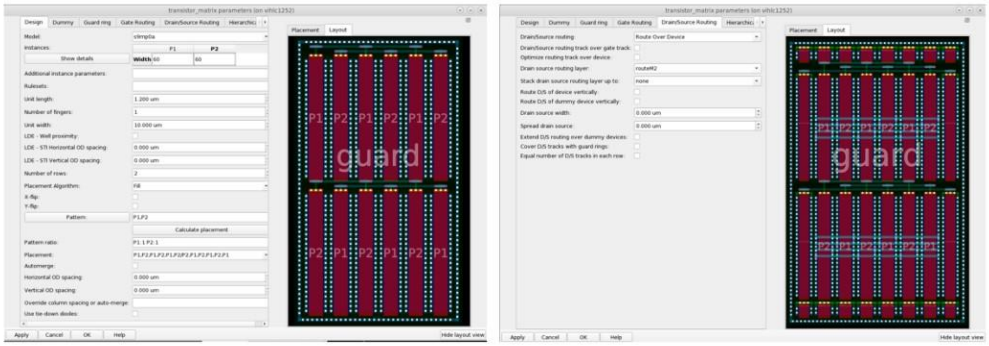

Figure 10: Qgen transistor matrix GUI

Figure 9 presents a possible result from the use of the Qgen tool through the use of a graphic user interface (GUI). Instead Figure 10 presents the GUI itself with some of the possible properties such as number of fingers, placement pattern, contacts position, routing, guard ring and more, that can be selected for the generation of a transistor matrix module with the relative layout view.

1.3.1 Layout generator structure

Thanks to the introduction of a hierarchical approach, it is possible for a designer to code a generator by using as atomic cells the already existing generators inside the classical Qgen. Precisely two steps need to be followed. Firstly, through the GUI in a layout XL view a template of the generator has to be created. Secondly, it has to be imported in Qgen for the

creation of a Qgen project with the relative implementation files inside the VSCode IDE[3]. It is important to highlight a useful feature of Qgen: it offers scripts to generate most of the needed code for the user, hiding a lot of the procedural complexity present during the process.

Through a graphical view of Qgen project's structure, as presented in Figure 11, plus a description of each file created inside the project itself, we can easily understand how it works.

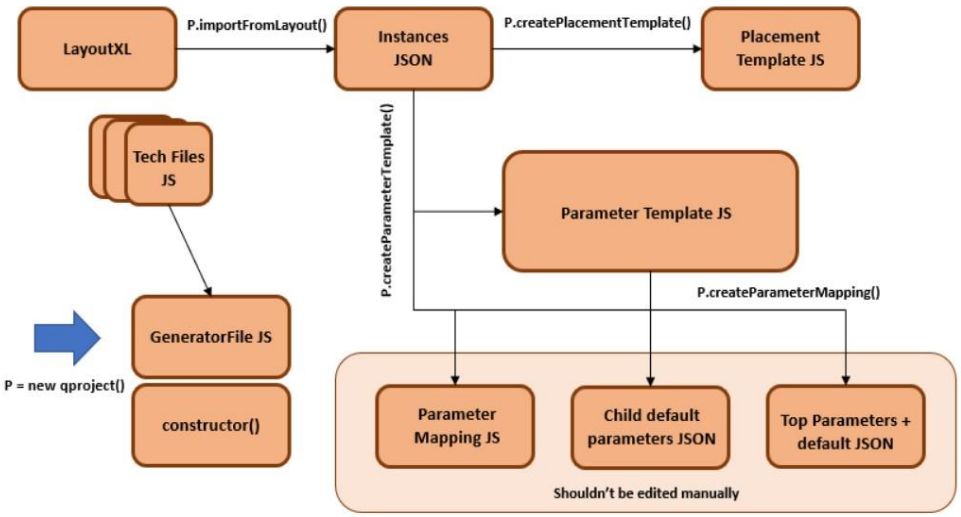

Figure 11: Qgen project structure

[3] Visual Studio Code is an integrated development environment made by Microsoft for Windows, Linux and macOS. Features include support for debugging, syntax highlighting, intelligent code completion, snippets and code refactoring

The procedure starts with the connection, through a telnet session, to a JavaScript (JS) terminal in which it is possible the creation of a project with two commands: the selection of the technology and the actual constructor call

```
init tech_Name

P = new qproject("project_Name","create","importFromLayout")
```

Listing 1.1: Commands to start a Qgen project

which automatically create the presented files following the hierarchical relations.

- Tech Files JS: these files are essential in order to make the generator work with the correct rules of the chosen technology such as number of metal available, DRC rules, particular layers and more.

- GeneratorFile JS: it is practically the core of the process which allows the creation of the Qgen project itself with all the other files

- Instances JSON: it embeds all the properties of the basic generators which are part of the imported layout view, and all this information is basically the one written in the GUI

- Placement Template JS: it contains all the information about the figGroups created in the layout view, plus the relative position of the basic module generators ordered in a graph structure

- Parameter Template JS: this is where the designer has to code the relation between the parameters that are considered relevant for the generator and the ones that are present in the Instances JSON file.

Every time that there are some changes inside the Parameter Template JS or in the Instances JS the remaining files are overwritten. Their functions are related to topics other

than those on which this thesis is focused, so they will not be discussed. As far as the routing is concerned, all the basic generators (e.g. transistor matrix) are routed individually at module level through the Qgen GUI, as can be seen in Figure 1.10, while routing between two different instances is made at higher hierarchical level thanks to a generator that produces tracks. Qgen is able to connect the modules through nets name and relationships, defined by the user, thanks to the track generator.

These concepts will be better explained in the dedicated layout section with the chosen methodology. What a designer needs to know before starting coding a layout generator with Qgen tool are the hierarchical structure and the one of the Qgen module generator itself.

CHAPTER 2

SCHEMATIC GENERATOR

This chapter presents the work done to code the driver schematic generator using the Anagen framework. First, a description of the test case and the followed methodology for the design is presented, as well as the various criteria that can be followed for the design of the driver. Along the chapter parts of the implemented code are presented, to provide a complete overview of the work performed.

2.1 Test case presentation

The test case of the thesis is the driver for a power switch of a dc-dc boost converter. The driver should be able to interface the logic, that generates the pulses coherently with what has been set by the control, with the power switch. The driver has to achieve the necessary capacity to drive a very large mos, and thus a very capacitive load, while making the logic see a much smaller load capacity, so as not to load it. The driver in the context just described must be able to achieve optimum timing performances while maintaining low power consumption and area occupancy.

This block is designed for the single power mos of a boost, but this does not preclude it from also being used for other converters, such as the buck dc-dc converter, by setting certain specs. This feature remarks on the possibility of reuse through the development of generators.

The study object in question is presented in the Figure 12.

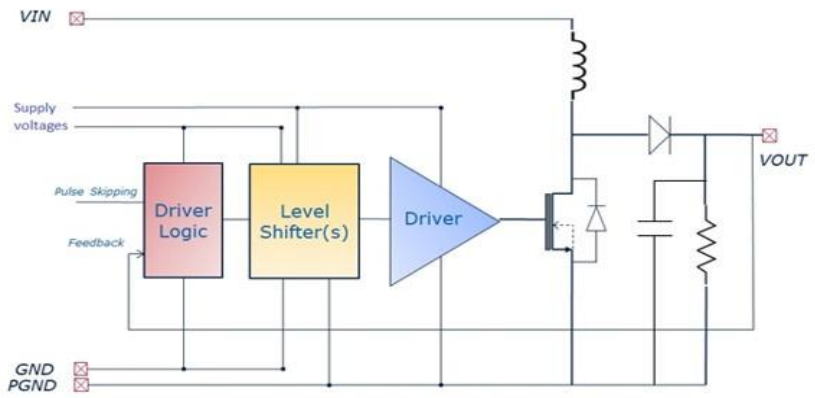

Figure 12: Test case presantation

In the following sections the driver block will be described in detail, analysed and the methodology for the development of the relative generators will be explained through design equations and the produced code.

2.2 Driver

As presented at the beginning of the chapter, the driver must be able to drive a high capacitive load, as the load consists of a large power nmos. The power nmos performs the function of a switch, so if a high signal is presented at the gate, it switches on, conversely if a low signal is presented, it switches off. The simplest driver that respects these characteristics and can be thought of is a CMOS inverter. In fact, it consists of two mos: a pmos which acts as a pull-up and provides the high signal at the output, and a nmos which acts as a pull-down and thus provides the low signal at the output.

However, the CMOS inverter alone would not be able to drive the high capacitive load, as there would be a degradation in timing performance. In particular a single inverter large enough to drive the power mos quickly would represent a considerable capacitive load for the logic driving the driver and, conversely, an inverter small enough to not load the logic would not be able to switch on the large power mos quickly enough. A cascade of inverters is therefore used to drive each mos of a strong-enough inverter, in order to meet the specifications, according to Figure 13.

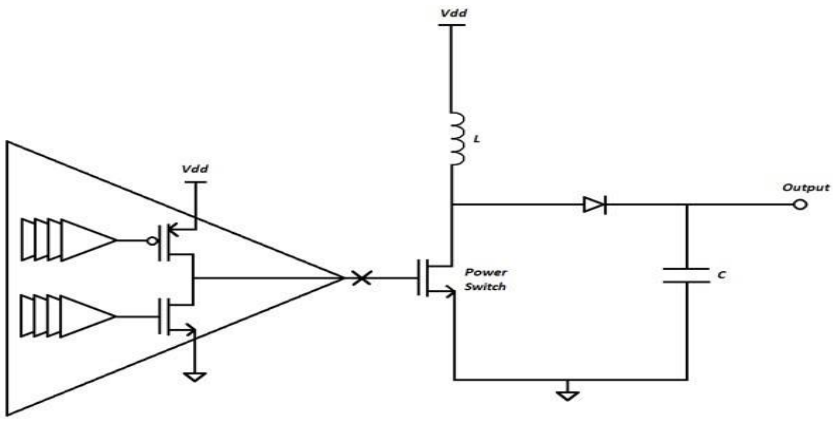

Figure 13: Internal driver structure

As can be seen in the Figure 13, there are two parts within the driver structure:

- the low-side part consisting of the driver cascade and the pull-down nmos
- the high-side part also consisting of a driver cascade and the pull-up pmos

It is useful to point out a typical problem that could occur with this configuration: cross-conduction, i.e. the time interval in which both mos of the last stage are switched on, causing a current draw from the power supply to ground and thus high consumption. The two driver cascades must be able to coordinate and avoid this problem. In this thesis, I have dealt with the design of the low-side part, which must be able to switch off the pull-down mos quickly (i.e. guarantee fast propagation of the switch-off signal) to ensure that there is no cross conduction at the moment when the pull-up pmos is switched on.

Now that this preamble on the driver structure has been made, it is clear that the starting point of the design and analogue generator is the choice of topology, that is presented in Figure 14.

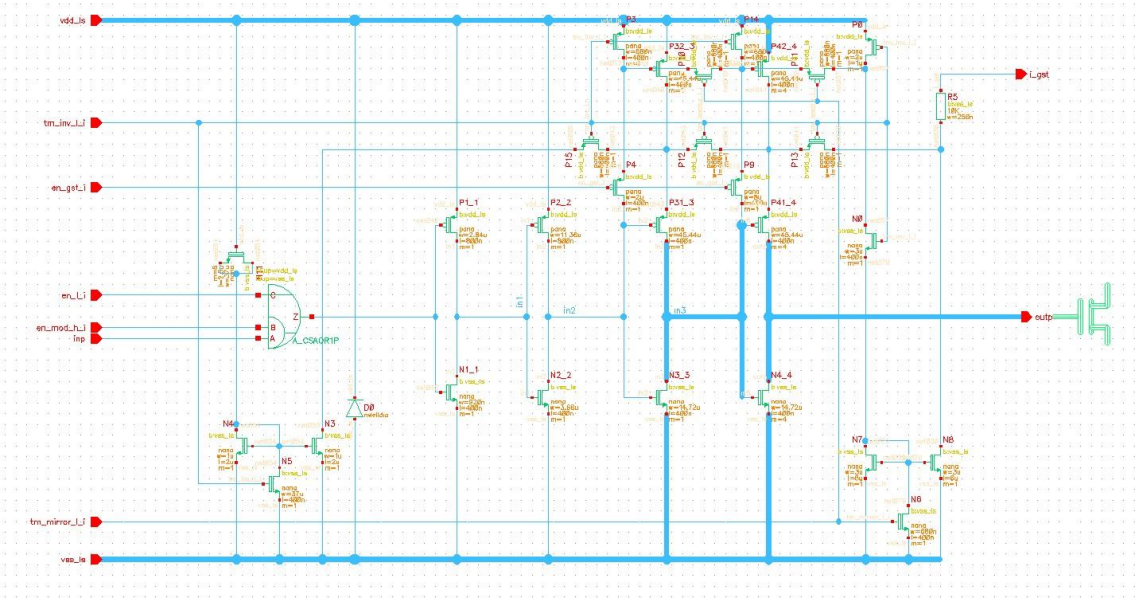

Figure 14: Driver topology

In our case, as can be seen in Figure 14, the starting topology consists of a cascade of CMOS inverters. Specifically, there are four inverter stages, which are highlighted below.

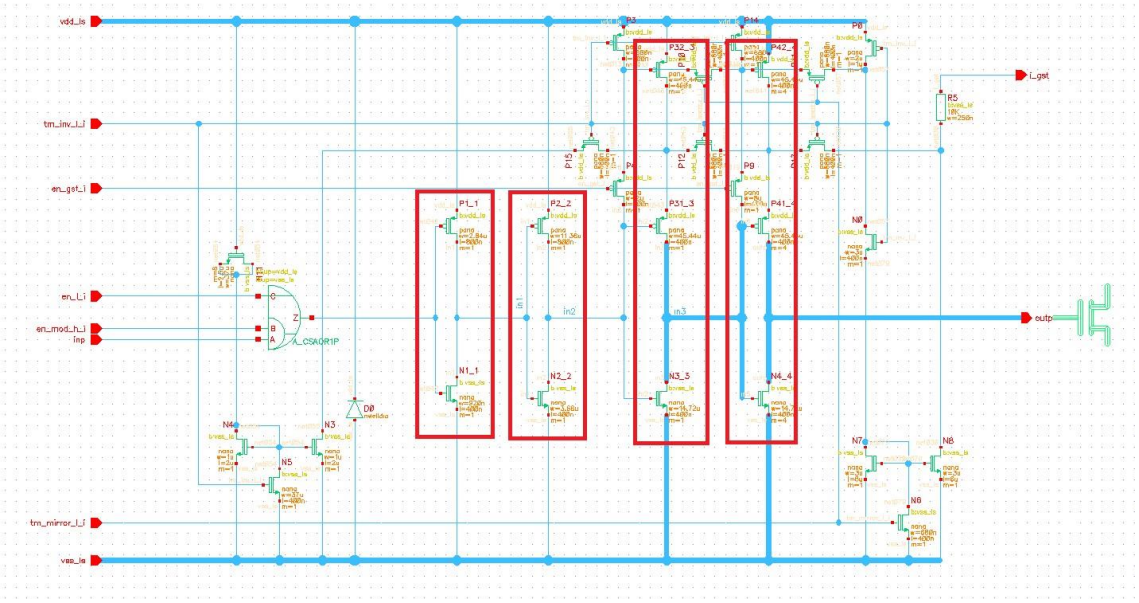

Figure 15: Driver topology with the stages highlighted

As can be note from Figure 15, if the first two stages consist of two simple CMOS inverters, the same cannot be said for the third and fourth stages: in particular, it can be seen that the pmos is not unique, but is split into two for both stages. The reason behind this split and the presence of several pmos on top, along with the two current mirrors, is to allow the circuit to be tested in various modes. In fact, when a driver is sized, in the final stages there are large MOS and there could be a reliability issue, i.e. you have to ensure that they function correctly and that there are no defects in the gate. Stress tests are done during production. In practice, some additional design is used to induce stress and make leakages measurements. We talk about Design for Stress DFS and Design for Testing DFT to indicate circuits that are not used for normal functionality, but during test and stress modes. These modes will not be dealt with in the thesis.

The idea in dimensioning a driver with this topology is to increase the size of each stage with respect to the previous one in order to allow the driving of a high output capacity while at the same time having good time performance.

There are various criteria for sizing a driver with this topology, which can be grouped into three macro approaches [5]:

- Fixed Taper Factor
- Variable Taper Factor
- Mixed Fixed and Variable Taper

As can be guessed, this factor plays an important role in dimensioning. When we speak of a fixed factor, there are various dimensioning criterion depending on whether we want to optimise the number of stages, to minimise the overall signal delay or minimise switching losses, or choose a dimensioning that is a trade-off between speed, occupied area and power consumption.

For this thesis, it was decided to implement the trade-off solution, since the topology remains fixed in the implementation of the schematic generator.

Due to this choice the taper factor alpha can vary in a range (3 : 6), while the number of stages is given by the following formula:

$$N = \log_{\alpha}(\frac{C_{out}}{C_G}) \tag{2.1}$$

where C_{out} is the capacitance to be driven and C_G is the input capacitance. Assuming one inverter has to drive an identical inverter as in Figure 16, the delay is dependent on the time constant, i.e. the product of its on-resistance with the total capacitance to be driven connected to its output; furthermore, the delay between input and output of the single stage is given by the time interval from when the input signal reaches 50% of its final value and the output signal does likewise, as represented in the same Figure 16.

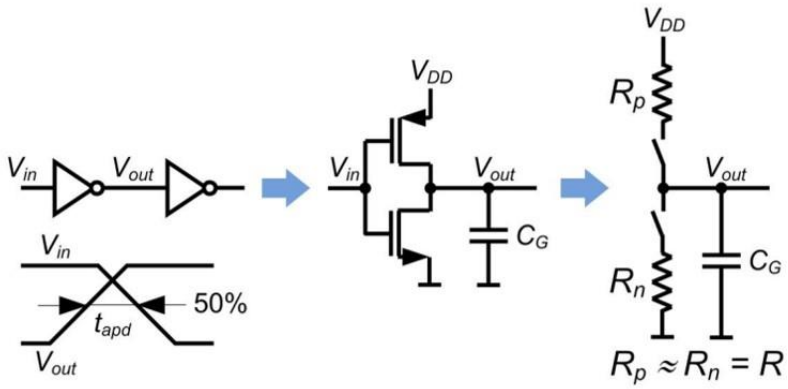

Figure 16: Inverter single stage

Assuming the capacitance to be driven is charged and the input signal transits from low to high, then this means that the output capacitance is discharging according to the law:

$$V_c(t) = V_s e^{\frac{-t}{R_{on} C_G}} \tag{2.2}$$

where V_s is the steady state voltage. Since we are concerned when the signal reaches half its value, i.e. when $V_c(t)$ is equal to $V_s/2$, substituting what has been said into the equation and solving it for t we obtain:

$$t_{pd} = \ln(2) * R_{on} C_G \tag{2.3}$$

where R_{on} is the on resistance, C_G is the input capacitance and finally t_{pd} is the propagation delay of a single stage. The same formula for the propagation delay can be reached assuming that the capacitance is discharged and the input signal transits from low to high. Thus, both resistance and the capacitance to drive play an important role in the formula. For the on resistance we have the following theoretical characterization:

$$R_{on} = \frac{1}{k'_n(\frac{W}{L}) * (V_{GS} - V_T)} \tag{2.4}$$

where V_T is the threshold voltage, V_{GS} is the gate source voltage, W and L width and length respectively, and K_n is the process-dependent parameter. Assuming that the stage to be driven is also an inverter, the capacity to be driven is proportional according to the following

$$C_G \propto W * L * C_{ox} \quad (2.5)$$

Now that there is a complete picture of the formulas, it is possible to extend the theory to the entire driver and in particular in the implementation of the fixed taper factor with trade-off solution.

The design criterion, that has been chosen, is to increase the size of each stage α times the previous one (by varying width, number of finger and multiplicity) as illustrated in the Figure 17.

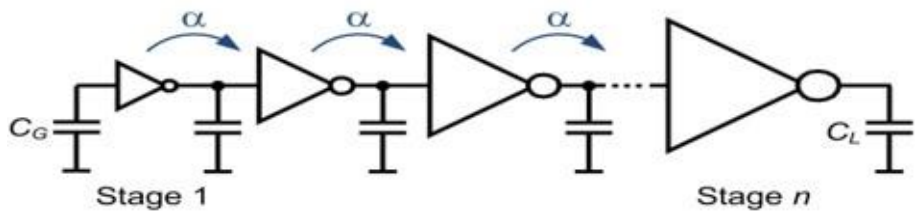

Figure 17: Driver stages

In particular, following this implementation and observing that the on-resistance is inversely proportional to the width, thus decreasing with alpha, while the capacitance being directly proportional always with the width, increases with alpha, we have that the product *RC*, from a theoretical point of view remains constant, thus the overall delay of this solution is given by the following formula:

$$t_{cascade} = N * \alpha * t_{pd} = N * \alpha * \ln(2) * R_{on}C_G \quad (2.6)$$

where N is the number of stages, α is the tapered factor, R_{on} is the equivalent on resistance of the first stage and C_G is the input capacitance of the cascade in Figure 17.

Since in the case of the thesis the driver topology is fixed, so the number of stages is four, it is straightforward to find the value of alpha to be used for dimensioning by inverting the formula 2.1 and obtaining

$$\alpha = \sqrt[N]{\frac{C_{out}}{C_G}} \quad (2.7)$$

However, a clarification must be made regarding the circuit sizing. Firstly, the topology of the driver does not present the cascade of simple CMOS inverters, but rather, as already noted, the third and fourth stages have the pmos split in two series device; moreover, in each of these two stages, the two split pmos have gates connected via another pmos. These are highlighted in the Figure 18 and acts as switches.

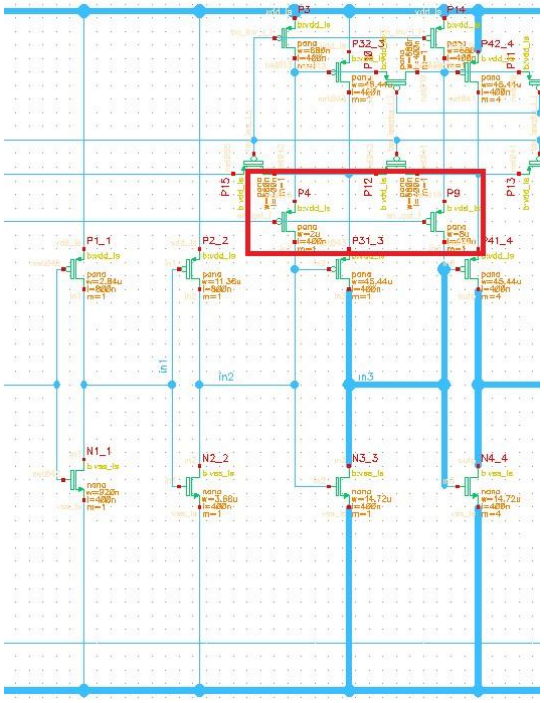

Figure 18: Pmoses as switches

The gate signal of P4 and P9 highlighted is always low, but the drain signal is subjected to a time-varying signal that causes a change of states: in particular P4 and P9 can fully switch off pmos P32_3 and P42_4, but not fully switch on them by bringing their gate to ground. In fact, when the gates of P32_3 and P42_4 go down (following the signal at nodes in2 and in3 respectively) to the threshold value of P4 and P9, these latter turn off and stop dragging down the gates of P32_3 and P42_4. Furthermore, it must be taken into account that P4 and P9 delay the switching on of P32_3 and P42 _4 compared to P31_3 and P41_4, as they transfer the signal with a delay due to their equivalent Ron. This behaviour leads to

an important variation in the intermediate performance of the driver, as a large on-resistance of the switch leads to a slowing down of the switching on of the second pmos and thus to a change in the overall on-resistance seen at the output with a consequent change in the delay of the stage in question. Thus, both switches become an active part in the sizing of the driver.

The second point in question is the capacitance, which no longer varies in the same way, i.e. it no longer increases with alpha. Let us assume that the two switches are short circuits to facilitate comprehension as in the Figure 19.

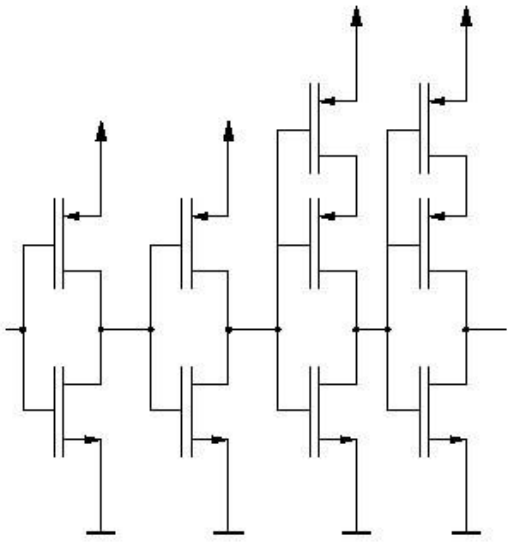

Figure 19: Simplified version

In order for the two pmos in series of the third and fourth stage to be equivalent to a single pmos of size *W/L* from the Ron's point of view, there are two choices: either double the width of each, or halve the length of each. The first option is not chosen, as this would imply an equivalent capacitance four times bigger, so the second option is chosen (keeping the equivalent capacitance unchanged) and the length has been halved in relation to the previous stages.

So far, we have seen the theory behind the implementation of the trade-off solution, but by using a fixed factor combined with some other criterion, it is possible to improve certain performances such as the case of asymetrical sizing, where a certain transition in the signal

propagation is speeded up. The idea behind the theory is to unbalance the MOS in the inverter cascade, as shown in the Figure 20.

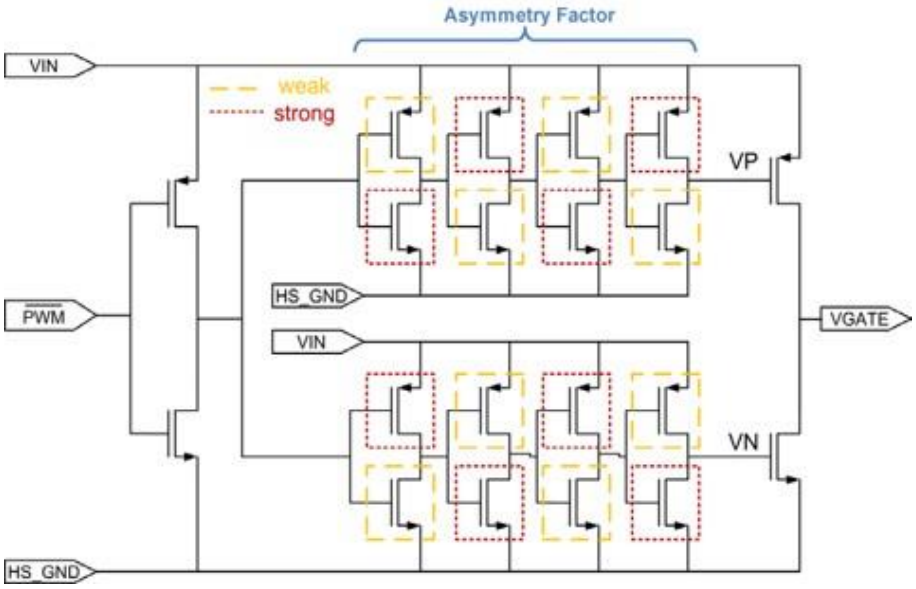

Figure 20: Asymmetrical sizing

As can be seen in Figure 20, in each stage of the cascade one mosfet is made stronger than the one previously evaluate, while the other is made weaker, which translated into sizing means making one mos larger and its complement smaller, keeping lenghts as they were. With this design choice, one transition is faster, while the other is slower; for example, if the pmos is sized larger and the corresponding nmos smaller, the high-to-low input transition will propagate faster than the low-to-high.The reverse is also true.

The stages are unbalanced with this formula:

$$W_{strong,weak} = W\,(1 + - \frac{\mathrm{AF}}{100\%}) \qquad (2.8)$$

where W is the width of the mos and AF is the asymmetric factor which is chosen in the range (20 : 30)%. This percentage range is chosen because a too low value involves a risk of not seeing any effect, while a too high percentage leads to speeding up a lot one transition, but also to making the other particularly slow.

2.2.1 Methodoloy

It has been explained the theory behind driver dimensioning. Now the first step is to characterise the driver, i.e. to obtain the necessary parameter measurements for dimensioning required by theory, then develop a methodology leading to the coding of the schematic generator.

The fundamental parameters for its dimensioning are firstly the dimensions of the first-stage mosfet, which must then have a symmetrical characteristic, typical of an inverter. Secondly, the driver input capacitance, dependent on the first stage, and the output load capacitance must be determined. Finally, the equivalent onresistance of the mosfets must be estimated.

For the first stage, the dimensions of the virtuoso library inverters were taken into account and tested whether the characteristic was symmetrical. In Figure 21 can be seen the choosen sizes for the pmos and the nmos, while in Figure 22 the characteristic.

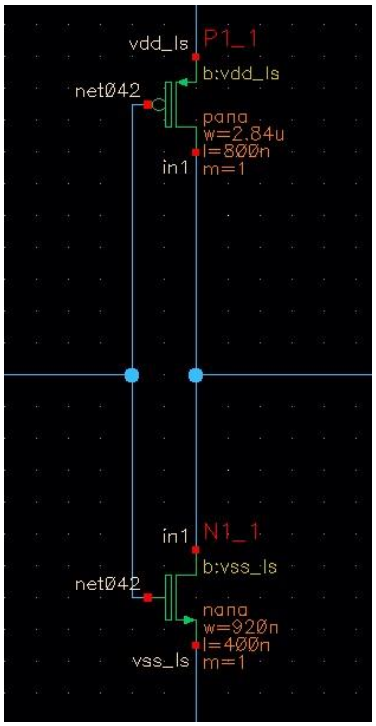

Figure 21: First stage dimension

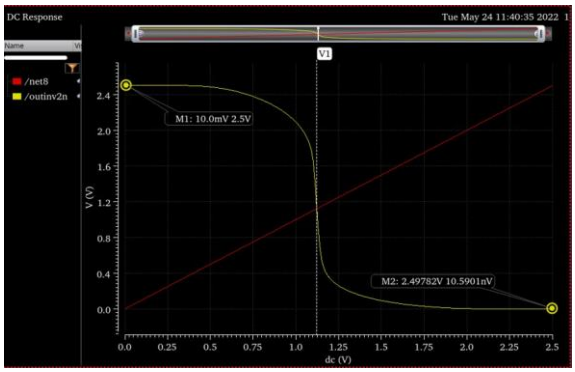

Figure 22: First stage characteristic

The choice of the size of the first stage also determines the input capacity of the driver, which must be calculated. In fact, the gate of the mosfet is essentially a capacitance and for the transistor to turn on, a current must be supplied. Both the pmos and the nmos are connected to the input node, and for the measurement of the capacitance it was decided to vary the voltage signal from low to high, measuring the integral of the current, namely the charge. The measurement configuration is presented in Figure 23.

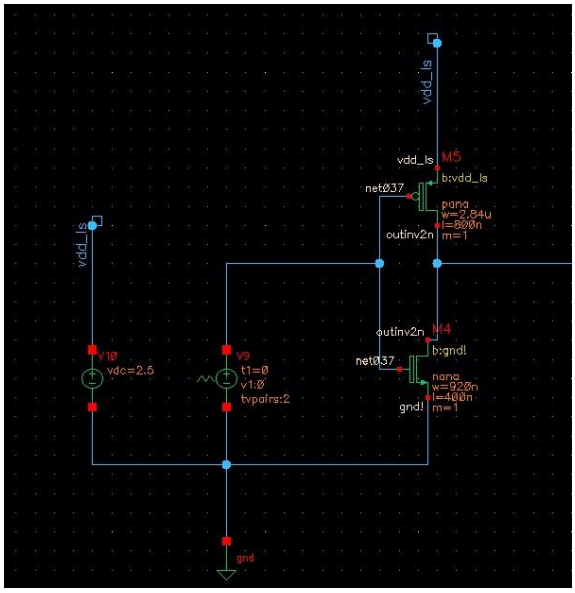

Figure 23: Input capacitance measure

A change in input voltage leads to a change in input current, so the following formula is used to calculate the input capacitance, with reference to Figure 24.

$$C_{in} = \frac{1}{\Delta V} * \int_{t0}^{t1} i(t)dt \tag{2.9}$$

where ΔV is the variation of the input voltage, and $i(t)$ is the current integrated.

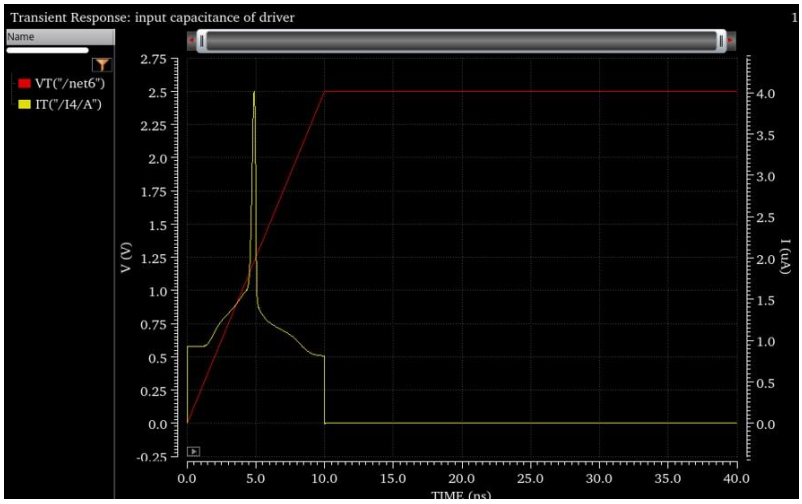

Figure 24: Input current absorbed

The measure obtained is C_{in} = 13,15fF. The same procedure was followed to calculate the output capacitance, where in this case there is the pull-down switch as in the Figure 25.

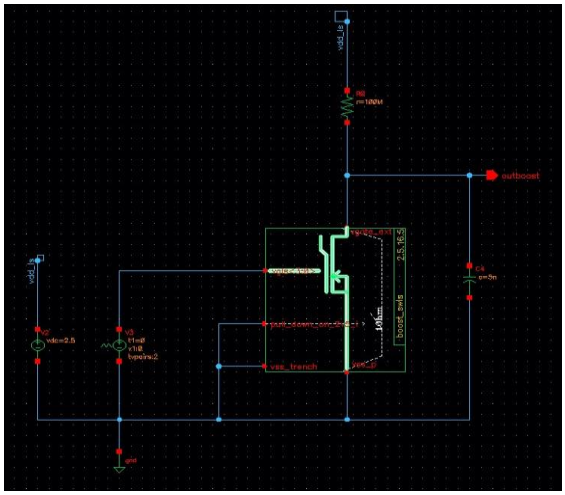

Figure 25: Configuration for output capacitance calculation

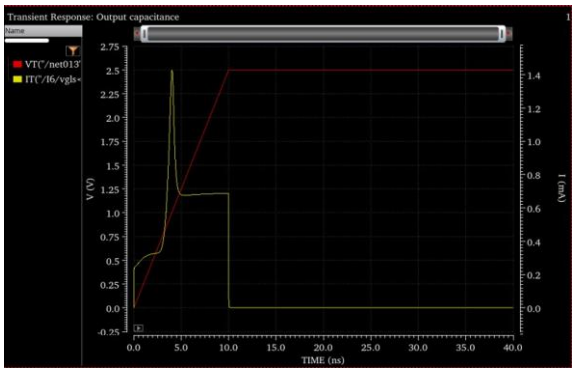

Figure 26: Output current absorbed

The measure of output capacitance is C_{out} = 2,9*pF*. As a final point, the value of the on-resistance must be characterized, which, as seen, it is a fundamental part of the theory. The value is calculated by taking the formula 2.6 and solving it for R_{on}:

$$R_{on} = \frac{t_{pd}}{\ln(2) * C_G} \tag{2.10}$$

At this point, a transient simulation of the circuit is performed, and the propagion delay for both the low-to-high and high-to-low signals, which determine the on resistors for the nmos and pmos respectively, is evaluated, and the following values are found: $R_{on,nmos}$ = 4986Ω e $R_{on,pmos}$ = 7842Ω. For the implementation of the code and as suggested by theory, the mean value is taken:

$$R_{on} = \frac{R_{on,nmos} + R_{on,pmos}}{2} = 6400\Omega \tag{2.11}$$

The parameters that determines the sizing of the driver are:

- Input capacitance C_G
- Output capacitance C_{out}
- Number of stages *N*
- Equivalent on resistance R_{on} according to the target parameter which is *total delay*

Now that the necessary values for the implementation of the generator have been found, the workflow that was followed for the implementation of the code is presented in Figure 27, while the devices that will be sized are reported in Listing 2.1 with the ANAGEN properties that are currently available:

- Transistor: width, length, multiplicity, number of finger and intent.

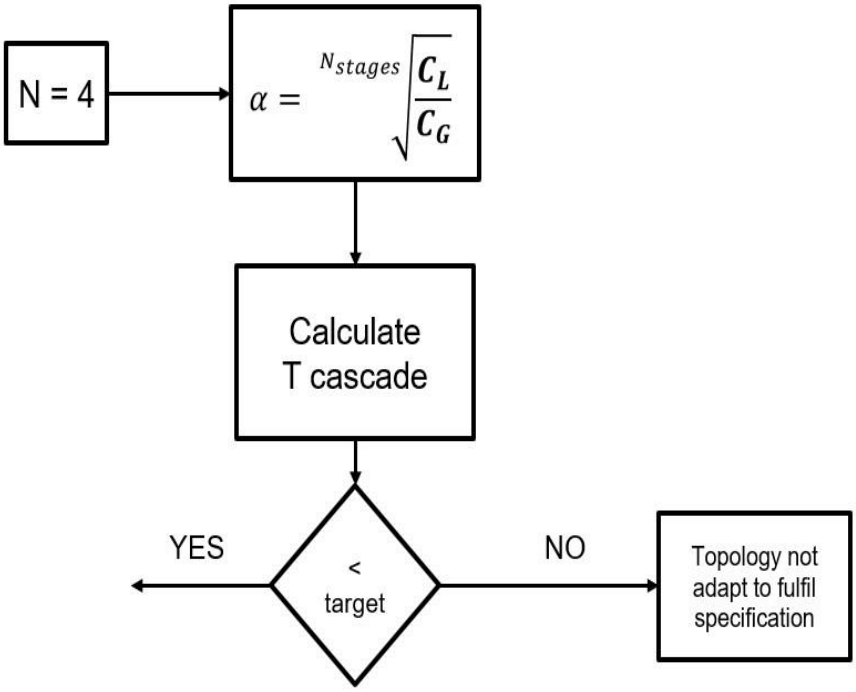

Figure 27: Workflow

```
{
**trans_specs("N1_1", "Driver_boost_ls", w = 0.92e-6, l=400n, multi
    =1, nf=1, intent= "nana")
**trans_specs("N2_2", "Driver_boost_ls", w = 0.92e-6, l=400n, multi
    =1, nf=1, intent= "nana")
**trans_specs("N3_3", "Driver_boost_ls", w = 0.92e-6, l=400n, multi
    =1, nf=1, intent= "nana")
**trans_specs("N4_4", "Driver_boost_ls", w = 0.92e-6, l=400n, multi
    =1, nf=1, intent= "nana")
**trans_specs("P1_1", "Driver_boost_ls", w = 2.84e-6, l=800n, multi
    =1, nf=1, intent= "pana")
**trans_specs("P2_2", "Driver_boost_ls", w = 2.84e-6, l=800n, multi
    =1, nf=1, intent= "pana")
**trans_specs("P31_3", "Driver_boost_ls", w = 2.84e-6, l=400n, multi=1, nf=1, intent= "pana")
**trans_specs("P32_3", "Driver_boost_ls", w = 2.84e-6, l=400n, multi=1, nf=1, intent= "pana")
**trans_specs("P41_4", "Driver_boost_ls", w = 2.84e-6, l=400n, multi=1, nf=1, intent= "pana")
**trans_specs("P42_4", "Driver_boost_ls", w = 2.84e-6, l=400n, multi=1, nf=1, intent= "pana")
**trans_specs("P9", "Driver_boost_ls", w = 2e-6, l=400n, multi=1, nf=1, intent= "pana")
}
```

Listing 2.1: List of the devices to be sized

Unlike the old version of ANAGEN, where devices were mapped from the schematic with a given name to a different name in ANAGEN environment, and then provided a method in which all had to be declared, now with the final version of the framework, the devices that are recognised by the technology are directly defined and present in the *self* instance of the generating class, the heart of the python file.

So, knowing that to access the individual devices concerned, one only needs to access the self instance, all that remains is to define the initial paremetres characterised above, i.e. the *sizing parms* in Listing 2.2.

```
def     get_default_specs(self)->dict:

            """ Returns the default specifications of this
            generator"""

                    specs = super().get_default_specs()

                                                                    # Get the default specs

                    specs["sizing_params"].update(                  #   modify the specs

                    {

                            "c_in" : 13.15e-15,

                            "c_load" : 2.892e-12,

                            "resistance_on" : 6400,

                            "number_stages" : 4                 #
                            Number of stages is fixed due to the
                            schematic

                            “tapered _factor” : 1,                  #
                            Value to calculate in order to meet the
                            specifications

                            “AF” : 20,

                            “high_level_params” :

                                    {    “delay_max” : 1.5e-9    }

                            }

                                                                        }

                            ) return specs              #return the specs
```

Listing 2.2: initial condition method

We then implement the method that provides driver dimensioning following the trade-off solution. As explained above, this consists of making each stage larger than the previous one by a certain taper factor.

However, indiscriminately increasing the width of the mos or the multiplicity (e.g. by a simple factor of 5, the multiplicity can achieve for the last stage a value of 125) leads to problems in terms of layout, specifically the area occupied, as well as

significant parasitic effects.

In fact, having transistor models with a high W/L can lead to problems, as well as not being electrically modelled well.

In addition, when the matching of several devices is relevant, a greater precision is got by splitting each MOS in a certain number of devices in parallel and placing them according to a particular layout structures.

Therefore, in the implementation of the method, a criterions is needed on how to distribute the taper factor intelligently, so that it then helps when implementing the layout generator. The criterion chosen to distribute the alpha factor is as follows.

Driver 2 Stage:

- width of the NMOS α times greater than the first stage
- width of the PMOS α times greater than the first stage

Driver 3 Stage:

PMOS splitted in two PMOS, so the length of each is the half

- width and number of finger of the NMOS α times greater than the second
- width and number of finger of the PMOS α times greater than the second

Driver 4 Stage:

- Multeplicity of the NMOS α times greater than the third stage
- Multeplicity of the PMOS α times greater than the third stage
- Switch $P9$ is α times greater than the switch on the third stage

In listing 2.3 there is the implementation of the method that sizes the driver according to the criterion explained.

```
# Method to size the Driver following the trade off solution def
driver_tradeoff_sizing(self, specs):

    # Specification needed to design the driver c_in = float(specs["c_in"]) c_load =
    float(specs["c_load"]) number_stages = float(specs["number_stages"]) r_on =
    float(specs["resistance_on"]) target_delay =
    float(specs["high_level_params"]["delay_max"])

    # Calculation to evaluate the number of stages capacitance_ratio =
    c_load/c_in tapered_factor = pow(capacitance_ratio, 1/number_stages)
    specs["tapered_factor"] = tapered_factor

    # Evaluation of a single propagation delay and the total delay with the starting
    condition


 tpd = 0.69 * r_on * c_in      # ln(2)= 0.69 is needed for the calculation of the total
delay    total_delay = number_stages * round(tapered_factor) * tpd

 # Requirements verification if total_delay
 <= target_delay:
```

```
        print("\n ---------- \u001b[36mSpecifications satisfied\u001b[0m ----------\n")

        else:

         print("\n---------- \u001b[31mSpecifications NOT satisfied\u001b[0m ----------\n")
 # Case of optimal solution that can be obtain, i.e. the tapter factor is a value +- 5% of Euler numberm and number of stages coincide with 4, the fixed one

 if number_stages == round(math.log(capacitance_ratio, tapered_factor)) and tapered_factor > 2.58 and tapered_factor <

2.85:

        print("\n\n\u001b[36m[INFO]\u001b[0m " + f"The the tapered reached a value between +-5% of the Euler number, which is the best solution to minimise the overall delay.")

 # Sizing 2 by increasing the width through the tapered factor
 self.sch_gen.set_ti_parameter("N2_2", "w", float(self.sch_gen.

instances["N1_1"]["parameters"]["w"]["value"]) * round( tapered_factor))
self.sch_gen.set_ti_parameter("P2_2", "w",
float(self.sch_gen.instances["P1_1"]["parameters"]["w"]["value"]) * round( tapered_factor))

# Setting the width of the mosfets of 3 stage

self.sch_gen.set_ti_parameter("N3_3", "w",
float(self.sch_gen.instances["N2_2"]["parameters"]["w"]["value"])*
round( tapered_factor))
self.sch_gen.set_ti_parameter("N3_3", "nf", round( tapered_factor))

self.sch_gen.set_ti_parameter("P31_3", "w", float(self.sch_gen.

instances["P2_2"]["parameters"]["w"]["value"])*round( tapered_factor))
self.sch_gen.set_ti_parameter("P31_3", "l", float(self.sch_gen.
```

```
instances["P2_2"]["parameters"]["l"]["value"])/2)
  self.sch_gen.set_ti_parameter("P32_3", "w", float(self.sch_gen.

instances["P2_2"]["parameters"]["w"]["value"])*round( tapered_factor))
self.sch_gen.set_ti_parameter("P32_3", "l", float(self.sch_gen.

instances["P2_2"]["parameters"]["l"]["value"])/2)
self.sch_gen.set_ti_parameter("P31_3", "nf", round( tapered_factor))
self.sch_gen.set_ti_parameter("P32_3", "nf", round( tapered_factor))



# Setting the width of the mosfets of 4 stage
  number_finger = round(tapered_factor)



  self.sch_gen.set_ti_parameter("N4_4", "w", float(self.sch_gen.



instances["N2_2"]["parameters"]["w"]["value"]) * round( tapered_factor))
self.sch_gen.set_ti_parameter("N4_4", "multi", round( tapered_factor))
self.sch_gen.set_ti_parameter("N4_4", "nf", number_finger)

self.sch_gen.set_ti_parameter("P41_4", "w", float(self.sch_gen.

instances["P2_2"]["parameters"]["w"]["value"]) * round( tapered_factor))
self.sch_gen.set_ti_parameter("P41_4", "l", float(self.sch_gen.
```

Listing 2.3: Trade off sizing for the driver

In the method, in addition to sizing, there is a check whether the target delay specification is met by the theoretical calculation; it also checks whether given the input and calculation values, the condition is optimum from the point of view of the driver's propagation delay, i.e. when the increment factor coincides with *e*.

In addition to the implementation of the trade-off method, another method is made available which, given the previous scaling, improves a user-specified transition following the previously explained criterion, that is the asymmetrical sizing and it is presented in listing 2.4.

```
# Method to size the driver to speed up one transition def asymetrical_sizing(self,
high_low, low_high):

      # Calculation of the factor to get strong/weak mos


  AF = float(self.sizing_params["AF"]) sizing_factor_weak = 1 - AF/100
sizing_factor_strong = 1 + AF/100

# Case to size for speeding up high to low transition

# In this case, in the final stage the nmos should be stronger than pmos

# Starting from this point, in an alternate way, the mosfets are sized

if high_low == 1 and low_high == 0:

# Sizing the Nmos in alternate way

self.sch_gen.set_ti_parameter("N1_1", "w", round(float(self
.sch_gen.instances["N1_1"]["parameters"]["w"]["value"]) * sizing_factor_weak, 10))
self.sch_gen.set_ti_parameter("N2_2", "w", round(float(self
.sch_gen.instances["N2_2"]["parameters"]["w"]["value"]) * sizing_factor_strong, 10))
self.sch_gen.set_ti_parameter("N3_3", "w", round(float(self
```

```
.sch_gen.instances["N3_3"]["parameters"]["w"]["value"]) * sizing_factor_weak, 10))
self.sch_gen.set_ti_parameter("N4_4", "w", round(float(self
.sch_gen.instances["N4_4"]["parameters"]["w"]["value"]) * sizing_factor_strong, 10))

        # Sizing the Pmos in alternate way

self.sch_gen.set_ti_parameter("P1_1", "w", round(float(self
.sch_gen.instances["P1_1"]["parameters"]["w"]["value"]) * sizing_factor_strong, 10))
self.sch_gen.set_ti_parameter("P2_2", "w", round(float(self
.sch_gen.instances["P2_2"]["parameters"]["w"]["value"]) * sizing_factor_weak, 10))

self.sch_gen.set_ti_parameter("P31_3", "w", round(float(
self.sch_gen.instances["P31_3"]["parameters"]["w"]["value"]) * sizing_factor_strong, 10))



self.sch_gen.set_ti_parameter("P32_3", "w", round(float(
self.sch_gen.instances["P32_3"]["parameters"]["w"]["value"]) * sizing_factor_strong, 10))
self.sch_gen.set_ti_parameter("P41_4", "w", round(float(
self.sch_gen.instances["P41_4"]["parameters"]["w"]["value"]) * sizing_factor_weak, 10))

self.sch_gen.set_ti_parameter("P42_4", "w", round(float(
self.sch_gen.instances["P42_4"]["parameters"]["w"]["value"]) * sizing_factor_weak, 10))



    return None
```

Listing 2.4: asymmetrical sizing method for the driver

The results obtained from the code will be presented in chapter four, together with the actual non-theoretical performance.

CHAPTER 3

LAYOUT GENERATOR

This chapter presents a briefly introduction of different approaches that can be followed for the layout generator. Then it proceeds with the discussion of the template choose for the implementation of the driver to end up with the use of Qgen, in order to get the generator of the instance.

3.1 Analog layout strategies

The strategies to develop a layout generator are different, but two macro categories can be distinguished according to the level of customization: full-custom or semicustom [6].

Full-custom allows a development with a great degree of freedom, leading to optimized desired performance. However, high effort is required unlike the semicustom, in which the effort decreases , but it leads to restrictions in development. The methods used to generate full-custom layouts are of interest and can be divided into optimization-driven and knowledge-driven semicustom [7].

Optimization-driven aims to automatically generate the layout through optimization algorithms based on cost functions. The purpose is to reduce the specified cost, for example the occupied area. The advantage of this approach is its generality, but on the other hand the disadvantage is the difficulty of setting up a cost function; in particular the complexity of this function is directly proportional to the problem and layout to be generated.

Knowledge-driven is an approach in which all the designer’s knowledge and expertise is codified. It is not as complex as the other, since the placement is already thought as a starting point. In this approach, layout rules or a predefined template can be followed during the development. In this thesis, a carefully designed template was set up and followed.

3.2 Driver Layout template

The starting point to develop a layout generator is the template that must be identified. For this thesis, the slicing floorplan was chosen for the template, as this particular structure allows the implementation of layout optimization algorithms, for example the Stockmayer algorithm, but which will not be discussed. However, the template chosen at the starting point allows future users to implement algorithms for placement optimization.

The slicing floorplan consists on arranging the blocks in such a way that the layout can be described by a binary tree, i.e. the total area is recursively divided until the leafs are reached, which represents the blocks, while the nodes have the information of how to group these leafs. In particular every node have the information about the cut that has to be made in the layout area to group blocks. For additional clarification of this structure, reference can be made to the Figure 28.

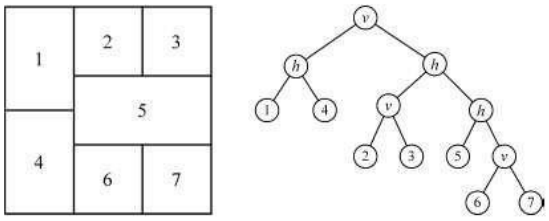

Figure 28: Slicing floorplan template and binary tree

The implementation of the slicing floorplan offers various advantages such as:

- It yields more compact layout instances
- Placement can be more easily specified by the relative positions of the layout tiles, since the hierarchy of slicing structures is better defined.
- It also allows to evaluate more easily other characteristics of the circuit layout such as routing

In this project driver sizing is of interest, hence the mosfets of the cascade stages.

However, in the starting topology, as can be seen in Figure 14, there are parts of the circuit that are required for stress and leakage measurements and that go beyond the topics discussed, but from a layout point of view they must be considered as they occupy an area. In particular the parts outside the cascade of inverters are those highlighted in the Figure 29.

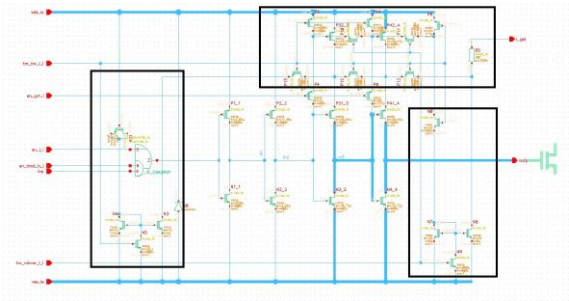

Figure 29: Circuitry around the Driver

Therefore, to make the template implementation clear and efficient, it was decided to group the external circuitry together and form three macro blocks:

- The capacitance which is the input nmos with drain and source short
- Nmos mirrors, input logic gate and resistance
- Pmos for measurements

Each macro-block can be considered as a single block in the template. Having made these preliminary considerations, the structure chosen to describe the layout is in Figure 30, where the driver is highlighted.

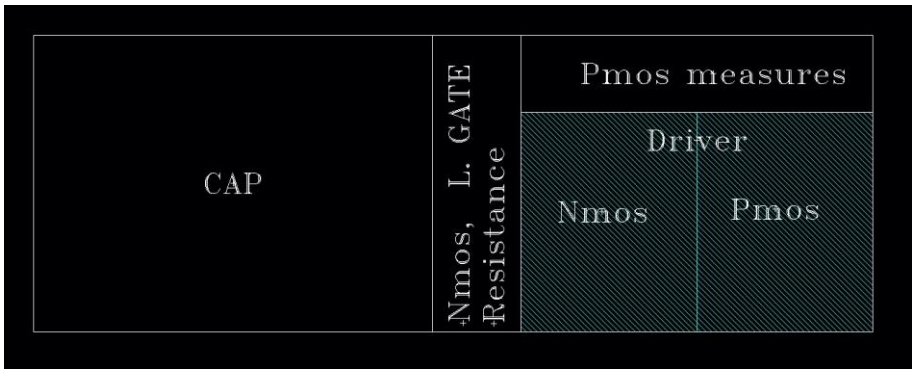

Figure 30: Template

The structure allows for a binary a tree description represented in Figure 31.

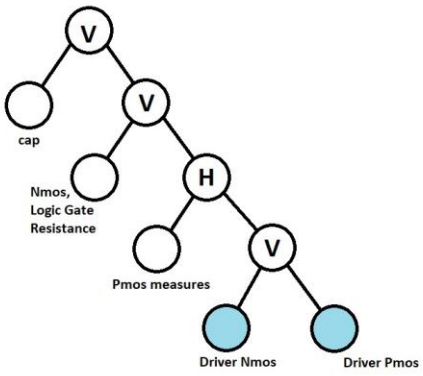

Figure 31: Binary tree description of the template

The implementation of such a template, where the position of individual blocks is well defined, allows the placement of traces used to connect adjacent and nonadjacent blocks. The limitation with the current version of Qgen is that traces are only horizontal or vertical. An example of a vertical trace and how it can connect blocks is presented in Figure 32.

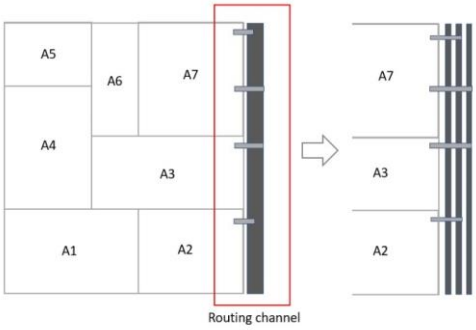

Figure 32: Routing channel example

Note that the structure in Figure 30 is a simplification, used as a starting point to be followed during the layout implementation.

In fact, the relative placement of the tracks has not been taken into account and macro-blocks have been considered, instead of the actual small blocks which are mosfets. As will be seen in the next section, the structure will become more complicated as Qgen itself implements its own tree following the user-made grouping.

In fact, for the current state of the tool, the internal algorithm developed consists of compacting the groups that the user creates, developing the placement tree. In this type of tree, each node can have multiple children and holds the information on how to compact the groups (vertically, horizontally or compact to lower left corner). However, other algorithms such as the previously presented are under development and will be integrated into the tool.

3.3 Qgen: Layout creation

The layout generator developed is the topology presented in Figure 14. As with the schematic generator a starting point is required: the template identified and implemented in layout xl view, available in the Cadence Virtuoso Design Environment, through the use of the Qgen GUI. The chosen template is presented in Figure 30.

All individual modules of the schematic are generated and positioned according to the template, so that they have a clearly defined position. The purpose of this approach is due to Qgen, which has an internal algorithm that compacts the blocks, following the relative positions through a graph structure.

In Figure 33 can be seen all individual modules that have been implemented through the Qgen interface, following the template previously proposed.

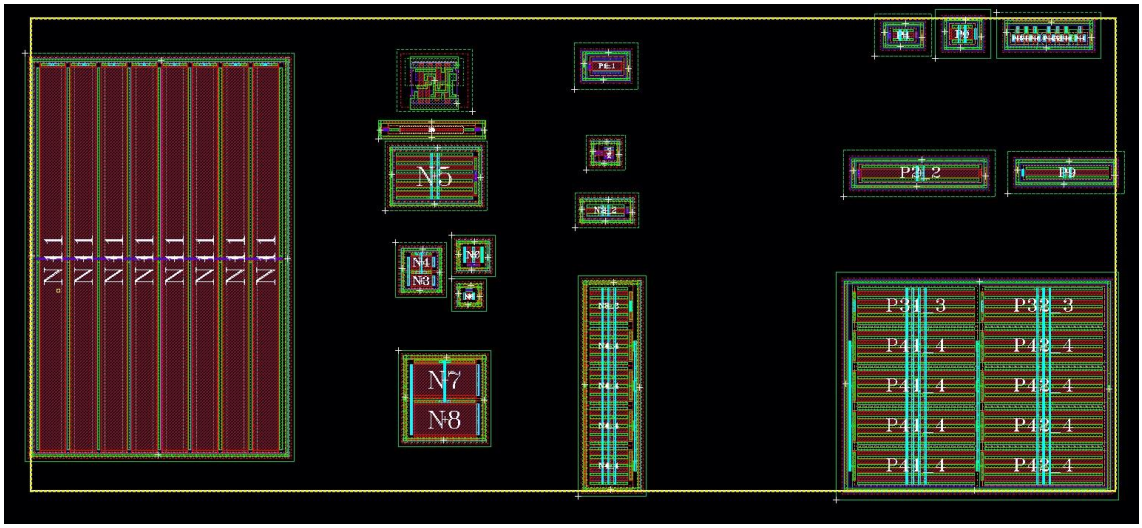

Figure 33: Modules implemented by Qgen

Qgen offers an important function: local routing. In fact, mosfets that are compatible in term of type and size (width or length) can be generated in a single block thanks to the regular matrix generator provided by the GUI. In this way internal routing can be set according to the user preferences. It can therefore be seen in Figure 33 that some of the

modules in the figure consist of several compatible mos and already have connections where possible.

By assembling the modules according to the chosen template, Qgen provides a tree description of the structure, presented in Figure 34. It can be seen that this description is complete and that each individual module in Figure 33 is a leaf of the tree.

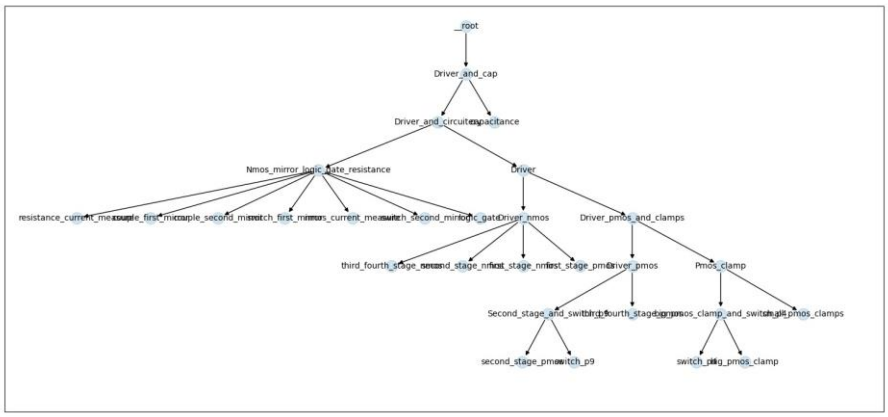

Figure 34: Tree description

The next step is to introduce the routing channels through a tracklines module generator. Once the channels are placed it is possible to use the Qgen import and connectivity editor presented in Figure 35 to construct again the hierarchical structure presented in Figure 34, but this time including also the relative position of the tracks inside the template. The new layout template is shown in Figure 36 while the new hierarchical structure is presented in Figure 37.

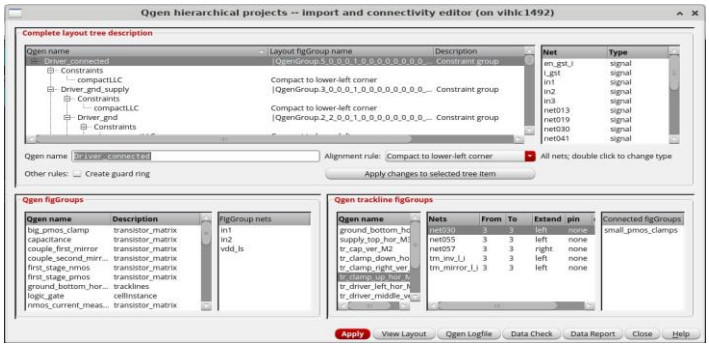

Figure 35: Import and connectivity editor

Figure 36: Template with modules and tracks

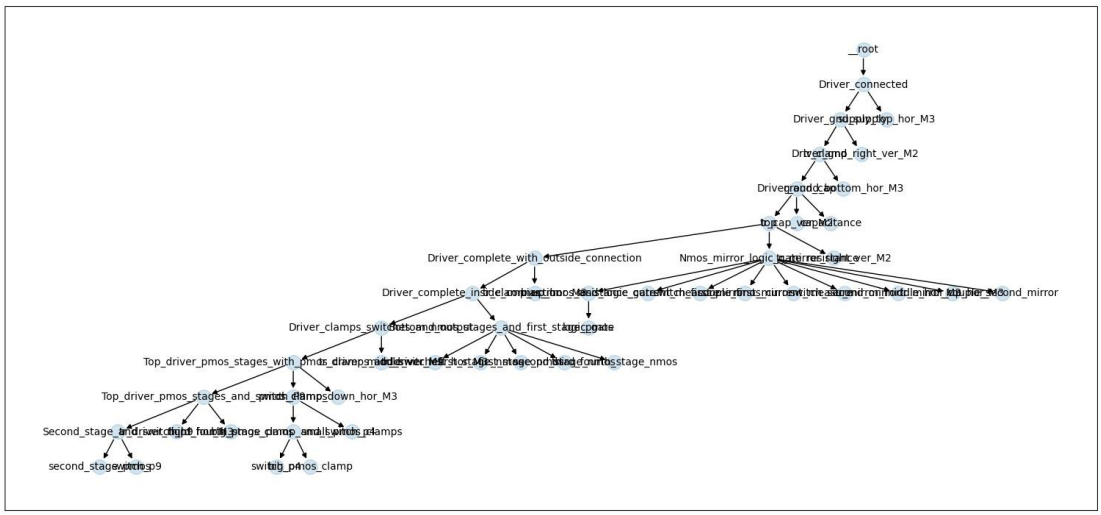

Figure 37: Hierarchical tree description

The main role of the import and connectivity editor in Figure 35 is to associate blocks-to-tracklines and tracklines-to-tracklines, in order to allow the built-in routing algorithm to connect two modules if the same net name is presented in the connectivity list of the modules themselves. This connectivity also assures that the tracklines are going to stretch with the modules associated to them. The limit of this routing algorithm is that it is not possible to connect two blocks directly without using a trackline and also the routing lines can only be vertical or horizontal without the possibility of alternative

shapes. This means that some area will be wasted, but the result is a complete routed layout which is DRC[4] clean.

Once the template is completed, the layout is imported inside a Qgen project with the commands presented in Listing 1.1, inside the VScode IDE through the telnet session. After this last phase, Qgen automatically creates the files (JavaScript) needed to develop the generator as already explained in the first chapter and in Figure 11. Specifically, in the Parameter Template JS file relations have been coded to manipulate the parameters of the template (width, length, number of finger etc...) and set the generator GUI. In particular the top level parameters have been set as editable variables, then functions which exploit parameters relations have been written in order to generate the correct layout instances. In the layout generator in question, the top level parameters made available to the user are:

- Nmos width of the first stage
- Pmos width of the first stage
- Tapered factor α
- Width for the switch P4

The four defined parameters are those that determine the sizing of the driver. In this way, there is compatibility between the schematic generator and the layout generator.

[4] Design rule checking: it is the verification of compliance with the design rules.

Then for the circuitry around the driver width finger, number of fingers and multiplicity are made editable.

Thanks to the Parameter Template JS file it is possible to modify the declared parameters when the Driver is selected from the list of Qgen generators, as can be seen in Figure 38.

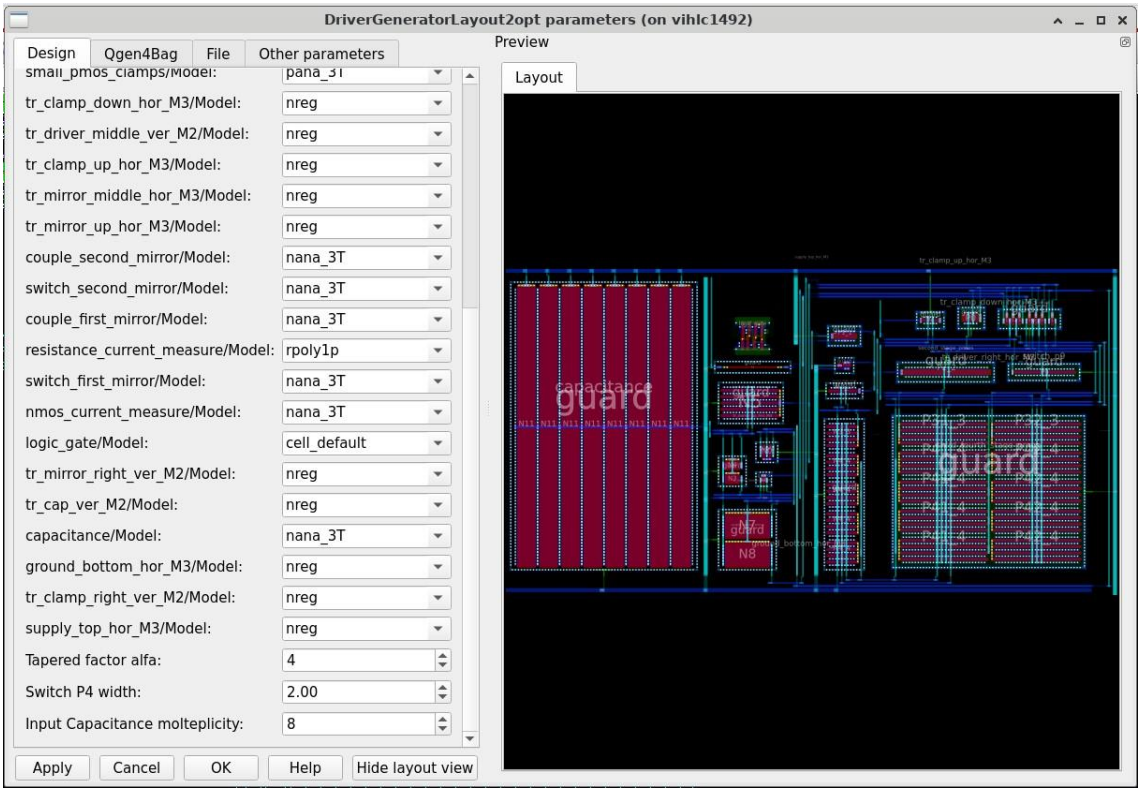

Figure 38: Driver generator GUI

The complete code of the layout generator is presented in Listing 3.1. At the beginning of the code there are all the modules instantiated for the layout; next, the top-level parameters and functions required are defined. Finally, there are sections in which parts of the driver are sized according to the desired criterion.

```
var childToParentParameterDefinition = function(instancesFile,
        Pd_to_Pc_MappingFile,Pc_DefsFile,Pd_File,Pd_DefsFile,

         parameterGraphFile,noGraph)

{

// Child instance: second_stage_nmos, generator:

     transistor_matrix

// Child instance: tr_driver_left_hor_M3, generator: tracklines

// Child instance: first_stage_pmos, generator: transistor_matrix // Child instance: first_stage_nmos,
generator: transistor_matrix

// Child instance: third_fourth_stage_nmos, generator:

     transistor_matrix

// Child instance: switch_p9, generator: transistor_matrix

// Child instance: second_stage_pmos, generator:

   transistor_matrix

// Child instance: tr_driver_right_hor_M3, generator: tracklines

// Child instance: third_fourth_stage_pmos, generator:

   transistor_matrix

// Child instance: switch_p4, generator: transistor_matrix // Child instance: big_pmos_clamp,
generator: transistor_matrix

// Child instance: small_pmos_clamps, generator:

   transistor_matrix

// Child instance: tr_clamp_down_hor_M3, generator: tracklines

// Child instance: tr_driver_middle_ver_M2, generator: tracklines

// Child instance: tr_clamp_up_hor_M3, generator: tracklines
```

```
// Child instance: couple_first_mirror, generator: transistor_matrix

// Child instance: couple_second_mirror, generator: transistor_matrix

// Child instance: switch_first_mirror, generator: transistor_matrix

// Child instance: switch_second_mirror, generator: transistor_matrix

// Child instance: nmos_current_measure, generator:
   transistor_matrix
// Child instance: tr_mirror_middle_hor_M3, generator: tracklines

// Child instance: tr_mirror_up_hor_M3, generator: tracklines

// Child instance: resistance_current_measure, generator:
   ResistorStack
// Child instance: logic_gate, generator: cellInstance

// Child instance: tr_mirror_right_ver_M2, generator: tracklines

// Child instance: capacitance, generator: transistor_matrix

// Child instance: tr_cap_ver_M2, generator: tracklines

// Child instance: ground_bottom_hor_M3, generator: tracklines

// Child instance: tr_clamp_right_ver_M2, generator: tracklines // Child instance:
supply_top_hor_M3, generator: tracklines

 var childParameters = LGenerator.prototype.getChildParameters( instancesFile);

 var PD = new PDatabase(childParameters);



// Declare new parameters here

PD.stringParameter('Qgen4Bag', 'Qgen4BagName', 'Qgen4Bag Name', '
   DriverGeneratorLayout2opt0');
```

```
//-----------------TOP LEVEL PARAMETER-------------------------

//Create a top-level parameter for the tapered factor obtain from schematic generator

// 4 = default value for the tapered factor

PD.integerParameter('Design', 'tapered_factor','Tapered factor alfa' , 4);

// Create a top-level parameter for "third_fourth_stage_nmos" and

    "third_fourth_stage_pmos"

// by editing "unitN" , "unitW"

// For a Transistor Matrix: "unitW" := "unitN" * "Wf"

// 03.68u = dafault valure for the finger width of the thirdfourth stage NMOS

// 11.36u = dafault valure for the finger width of the thirdfourth stage PMOS

PD.doubleParameter( 'Design', 'third_fourth_stage_Wf_nmos','Third fourth stage NMOS W_f', 3.680);

PD.doubleParameter( 'Design', 'third_fourth_stage_Wf_pmos','Third fourth stage PMOS W_f', 11.36);

// Create a top-level parameter for width first stage


PD.doubleParameter( 'Design', 'P4_width_switch','Switch P4 width', 2);


// Create a top-level paramenter for input capacitance

PD.integerParameter('Design', 'input_capacitance_M','Input Capacitance molteplicity', 8);


//-----------------FUNCTIONS-------------------------

// Define parameter relationships here
```

```
PD.childParameterOfInterest(['model']); // The child models must typically be dealt with

// Function that updates the number of finger variable var __NF_function = function(N,alfa)

{ return (N*alfa); }

// Function that updates the width variable var W_function =
function(Wf,N)

{ return (Wf*N); }

// Two functions for the molteplicity of the last stage (3 and 4) of the Driver

// Multeplicity is equal to the taper factor var __calcPlacement_third_fourth_stage_NMOS =
function (M)

{

    var P;

    P = calcPlacementPattern(["N4_4","N3_3"],[M,1],"fill"); return (P)

}

var __calcPlacement_third_fourth_stage_PMOS = function (M)

{

    var P;

     P = Array(M+1).join("P41_4*,") + "P31_3/" + Array(M+1).join("

     P42_4,") + "P32_3*" return (P)

}


// Function for the molteplicity of the capacitance var __calcPlacement_capacitance =
function(M)
```

```
{
    var P = calcPlacementPattern(["N11"],[M],"fill"); return (P)
}
// -----------------DRIVER-NMOS----------------------------
// Set the width of the second stage nmos from the userspecified tapered factor alfa
// unitW = ["second_stage_nmos","unitW"]
// unitW = tapered_factor * first_stage_Width_nmos --> 0.92 is the width of the first_stage nmos
PD.setParameterFunctionMI(__W_function, ["second_stage_nmos","
  unitW"],[["first_stage_nmos","unitW"],"tapered_factor"]);
// unitN = N --> With the function below I set the new number of finger specified by the user
PD.setParameterFunction(__NF_function, ["third_fourth_stage_nmos"
  ,"unitN"], "tapered_factor", [1]);
// Set the width for the third and fourth stage NMOS
// unitW = ["third_fourth_stage_nmos","unitW"]
// unitW = N*Wf = tapered_factor * third_fourth_stage_Wf_nmos
// Update the width finger according to the taper factor so in this way it match with the width of second
  stage
PD.setParameterFunctionMI(__W_function, [' third_fourth_stage_Wf_nmos'],
  [["first_stage_nmos","unitW"],' tapered_factor']);
PD.setParameterFunctionMI(__W_function, ["third_fourth_stage_nmos
  ","unitW"],["third_fourth_stage_Wf_nmos","tapered_factor"]);
// Update the molteplicity of the last stage according to the tapered factor
```

```
PD.setParameterFunction(__calcPlacement_third_fourth_stage_NMOS,
["third_fourth_stage_nmos","Placement"] , ["tapered_factor"]);



// ------------------DRIVER-PMOS-------------------------------

// Set the width of the second stage pmos from the user-specified tapered factor alfa

// unitW = ["second_stage_pmos","unitW"]

// unitW = tapered_factor * first_stage_Width_pmos --> 2.84 is the width of the first_stage pmos

PD.setParameterFunctionMI(__W_function, ["second_stage_pmos","
  unitW"],[["first_stage_pmos","unitW"],"tapered_factor"]);

// unitN = N --> With the function below I set the new number of finger specified by the user

PD.setParameterFunction(__NF_function, ["third_fourth_stage_pmos"
  ,"unitN"],"tapered_factor",[1]);

// Set the width for the third and fourth stage PMOS

// unitW = ["third_fourth_stage_pmos","unitW"]

// unitW = = N*Wf = tapered_factor * third_fourth_stage_Wf_pmos

// Update the width finger according to the taper factor so in this way it match with the width of second
  stage

PD.setParameterFunctionMI(__NF_function, [' third_fourth_stage_Wf_pmos'],
  [["first_stage_pmos","unitW"],' tapered_factor']);

PD.setParameterFunctionMI(__W_function, ["third_fourth_stage_pmos
  ","unitW"],["third_fourth_stage_Wf_pmos","tapered_factor"]);

// Update the molteplicity of the last stage according to the tapered factor

PD.setParameterFunction(__calcPlacement_third_fourth_stage_PMOS,[
```

```
"third_fourth_stage_pmos","Placement"] , ["tapered_factor"]);




//----------------Switch P9----------------------------

PD.setParameterFunctionMI(__W_function, ["switch_p9","unitW"],["

  P4_width_switch","tapered_factor"]);

//----------------Input Capacitance---------------------

// Update the molteplicity of the capacitance

PD.setParameterFunction(__calcPlacement_capacitance,["capacitance

  ","Placement"] , "input_capacitance_M");

// Plot the parameter dependency graph if ( noGraph != true )

PD.plot(parameterGraphFile);

// Write the parent-to-child mapping function; return(

PD.parentToChildren('DriverGeneratorLayout2opt',

Pd_to_Pc_MappingFile, Pc_DefsFile, Pd_File, Pd_DefsFile) );

}
```

Listing 3.1: Layout Generator code

After the layout has been instantiated, the connections that are not handled by Qgen in the current state of the tool are completed by hand. In the present case, it is only the connection of the logical port. Figure 39 shows the final result.

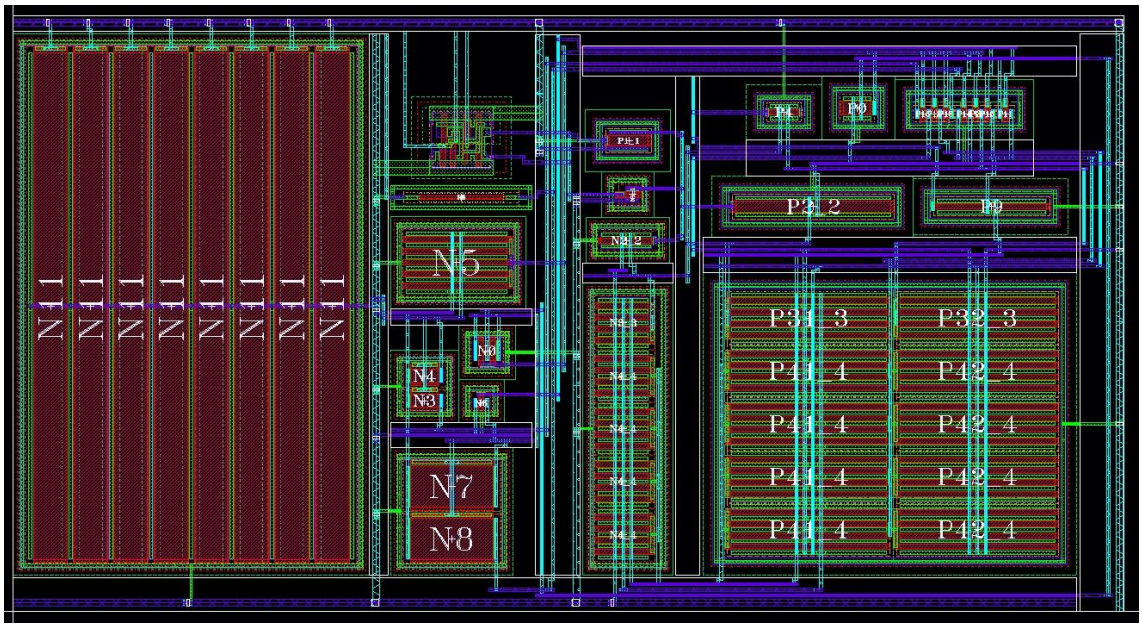

Figure 39: Final Layout

CHAPTER 4

RESULTS

This last chapter presents the results obtained from the schematic generator followed by the pre-layout verification, with discussion of the results. The complete flow of schematic generator and WiCked optimisation is presented. Finally, the result of the layout generator is introduced with discussion of the post-layout verification.

4.1 Schematic Generator flow

This section presents the schematic generator flow and how it interacts with the user, right up to the implementation of the instance. Then, the schematic was tested through simulations performed with Avenue.

The schematic generator code allows to take the specification from an external JAMA file. The specification for the code is the total delay from the input of the driver to the output. Nevertheless other measures have been evaluated and they will be discussed in the following section.

In Figure 40 it is possible to see how the code interfaces with the user.

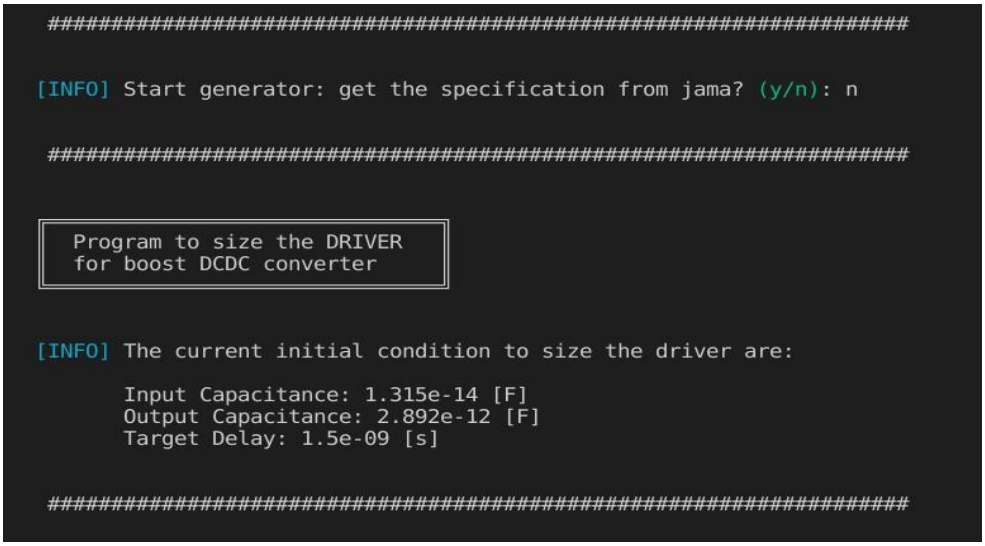

Figure 40: Starting the program

After deciding whether to take the specifications from Jama or to enter them manually, the code also offers the possibility of changing the parameters that determine the sizing of the driver, i.e. output capacitance, input capacitance, equivalent on-resistance and the size of the first stage. This step emphasises key features of the code which are re-usability and portability.

In the next step the generator code offers the choice of trade-off-sizing. In case of affirmative input, the code implements the criterion and sizes automatically the mosfets concerned for the Driver. A summary of the changes made in the schematic is provided to the user, which is shown in Figure 42.

```
[INFO] Mosfet dimension after trade-off solution:


List of NMOS
N1_1 --> width: 9.2e-07, length: 4e-07,  nf : 1.0,  multi: 1.0
N2_2 --> width: 3.68e-06, length: 4e-07,  nf : 1.0,  multi: 1.0
N3_3 --> width: 1.472e-05, length: 4e-07,  nf : 4.0,  multi: 1.0
N4_4 --> width: 1.472e-05, length: 4e-07,  nf : 4.0,  multi: 4.0


List of PMOS
P1_1  --> width: 2.84e-06, length: 8e-07,  nf : 1.0,  multi: 1.0
P2_2  --> width: 1.136e-05, length: 8e-07,  nf : 1.0,  multi: 1.0
P31_3 --> width: 4.544e-05, length: 4e-07,  nf : 4.0,  multi: 1.0
P32_3 --> width: 4.544e-05, length: 4e-07,  nf : 4.0,  multi: 1.0
P41_4 --> width: 4.544e-05, length: 4e-07,  nf : 4.0,  multi: 4.0
P42_4 --> width: 4.544e-05, length: 4e-07,  nf : 4.0,  multi: 4.0


List of PMOS as switch
P4 --> width: 2e-06, length: 4e-07,  nf : 1.0,  multi: 1.0
P9 --> width: 8e-06, length: 4e-07,  nf : 1.0,  multi: 1.0

[INFO] PMOS of 3° and 4° stage are split in two for this specific schematic
```

Figure 41: Mosfets sizing summary after trade-off implementation

The code also checks whether the theoretical delay meets the required specification and provides the growth value α used for the stages according to the circuit's initial conditions. An example of this result can be seen in Figure 42.

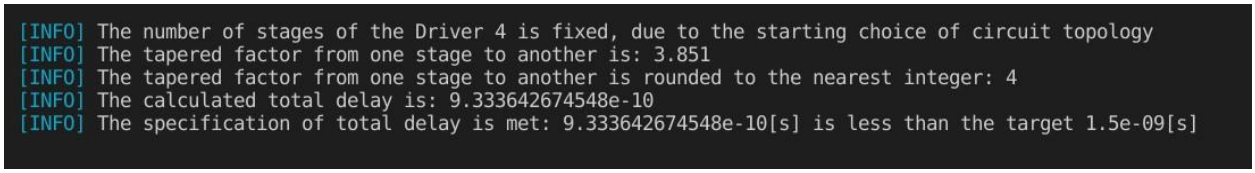

Figure 42: Requirement check

After the trade-off sizing, there is the possibility to decrease the time for a transition by using the asymmetrical factor. This interface can be seen in Figure 43.

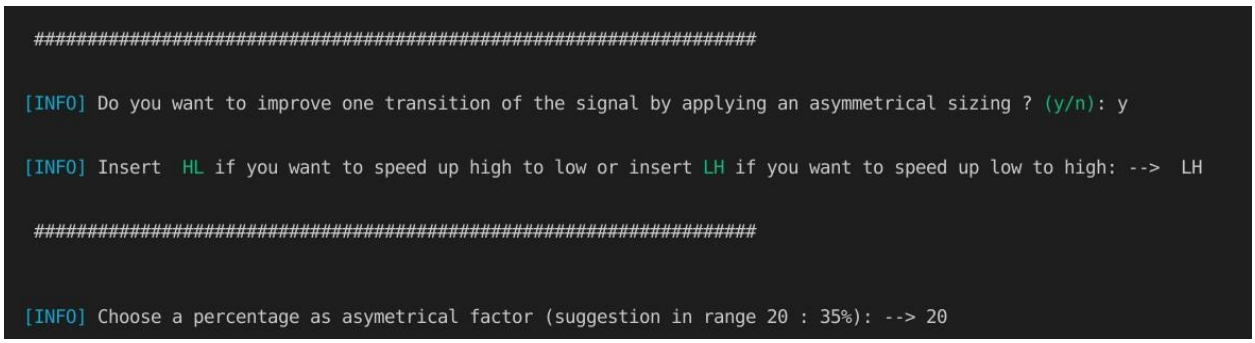

Figure 43: Asymmetrical sizing option

The code updates the values of the mosfets in the schematic according to the user interaction. For example, it can speed up the low-to-high transition by an asymmetric factor of 20% and the result is printed out as in Figure 44.

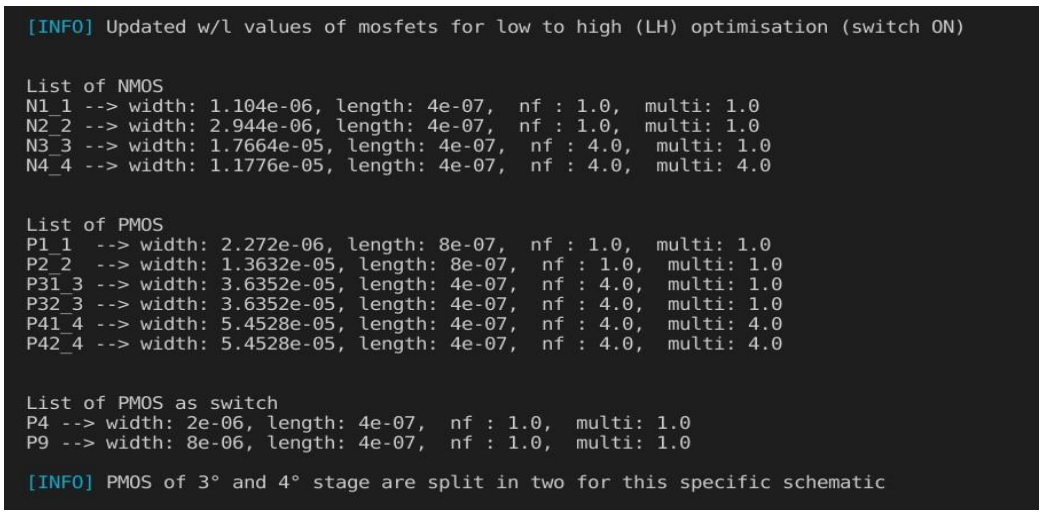

Figure 44: Asymmetrical sizing result

Asymmetrical sizing was implemented in the schematic generator to offer the user better transition if they were of primary importance. However, in the following sections, only the trade-off solution is analysed with discussion of the optimiser embedded in the flow.

4.1.1 Pre-Layout Verification

The schematic generator code instantiate a driver sized according to the implemented criterion. It is now necessary to verify the performances of the circuit through the evaluation of interested measures, that are:

- Delay on_driver ls: is the measure for the delay of the on signal, i.e. the propagation of the low-to-high transition
- Delay off_driver ls: is the measure for the delay of the off signal, i.e. the propagation of the high-to-low transition
- Rise lson: is the measure for the transition time (low to high) of the output node, better known as the rise time
- Fall lson: is the measure for the transition time (high to low) of the output node, better known as the fall time
- Curr_cons_vdd2v5: is the dynamic measure for the current consumption for devices supplied with 2.5V (Driver)
- Curr_cons_vdd1v5: is the dynamic measure for the current consumption for devices supplied with 1.5V
- Curr_leak_0B: is the static measure of leakage current assuming a fixed low input signal to the Driver

- Curr _leak 1B: is the static measure of leakage current assuming a fixed high input signal to the Driver

The simulation has been done using Avenue (Analog Verification Environment), which is an Infineon in-house tool that provides an environment to verify or manage transistor level simulation semicustom. Avenue allows to execute a verification plan related to the Driver and create a report. The needed test is run in batch mode by calling the *verify()* method, presented in Listing 4.1.

```
def verify():project_root = os.path.join(os.path.dirname(__file__), "../../ schematic/")

    generator= Generator_DRIVER_LS_BOOST_DRIVER_LS_TAPERED_BUFFER(project_root)

    specs = generator.get_default_specs() #generator.specs

    results_path = os.path.join(os.path.dirname(__file__), "../ avenue/results/") ave_path =
os.path.join(os.path.dirname(__file__), "../avenue/")

    os.makedirs(results_path, exist_ok=True) log.info(f"[INFO] Verification results are at: {results_path}")
    log.info(f"[INFO] Generating testbench and netlisting")

    save_path = os.path.join(os.path.dirname(__file__), "../results/spt9u/") netlist_path =
    os.path.join(os.path.dirname(__file__), "../ netlist/")

    if os.path.exists(netlist_path + "tb_driver.nd.tit"):

        os.remove(netlist_path + "tb_driver.nd.tit")

        print("[INFO] A previous version of the netlist was present and has been erased")

            options = '{"view": "schematic","netlist_dest": "' + netlist_path + "tb_driver.nd.tit"+ '"}'

            os.system("anagen_layver create netlist --lib " + generator.lib+ " --cell " + generator.cell + "

            --output " + "runATC.cfg" + "--options " + "'" + options + "'")
```

```
        os.system("anagen_layver run --lib " + generator.lib + " --cell" + generator.cell + " " +
        "runATC.cfg" + " --force") # -force: update the output folder   log.info(f"[INFO] Netlisting
        done!")
    log.info(f"[INFO] Setup Avenue plan")
    plan = pyavenue.avePlan(ave_path + 'templates/Verification_plan_template.apf')
    plan.simulate()  plan.viewResults()                return specs
```

Listing 4.1: Verify method to launch Avenue

The goal is to analyse the measures and assess whether they are acceptable. The main analysis is made under nominal condition. The measures were tested also in corner conditions to observe variations experienced. Finally, a Monte Carlo analysis is also performed, in which changes in measurements are observed not only in corner conditions, but also with changes in device parameters. The three tests are presented as follows.

TEST 1

measure	unit	description	specMin	measMin	measMax	specMax
Rise_Ison	s	Ison rise time	undef	713.7p	713.7p	undef
Fall_Ison	s	Ison fall time	undef	388.9p	388.9p	undef
Delay_on_driver_Is	s	Delay of the on signal low to high	undef	837.2p	837.2p	undef
Delay_off_driver_Is	s	Delay of the off signal high to low	undef	1.167n	1.167n	undef
Curr_cons_vdd2v5	A	Current consumption from the 2.5V domain	undef	27.87u	27.87u	undef
Curr_cons_vdd1v5	A	Current consumption from the 1.5V domain	undef	8.385n	8.385n	undef
Curr_leak_0b	A	Current leakage from 2.5V domain for input low	undef	2.238n	2.238n	undef
Curr_leak_1b	A	Current leakage from 2.5V domain for input high	undef	2.985n	2.985n	undef

Figure 45: Main measurements – typical conditions

TEST 2

measure	unit	description	specMin	measMin	measMax	specMax
Rise_Ison	s	Ison rise time	undef	308.5p	1.349n	undef
Fall_Ison	s	Ison fall time	undef	193.2p	881.4p	undef
Delay_on_driver_Is	s	Delay of the on signal low to high	undef	527.8p	1.565n	undef
Delay_off_driver_Is	s	Delay of the off signal high to low	undef	677.3p	2.228n	undef
Curr_cons_vdd2v5	A	Current consumption from the 2.5V domain	undef	18.04u	42.84u	undef
Curr_cons_vdd1v5	A	Current consumption from the 1.5V domain	undef	6.283n	219.2n	undef
Curr_leak_0b	A	Current leakage from 2.5V domain for input low	undef	158.1p	4.788u	undef
Curr_leak_1b	A	Current leakage from 2.5V domain for input high	undef	164.8p	6.108u	undef

Figure 46: Main measurements – corner conditions

TEST 3

measure	unit	description	specMin	measMin	measMax	specMax
Rise_Ison	s	Ison rise time	undef	341.2p	1.078n	undef
Fall_Ison	s	Ison fall time	undef	237.7p	706.3p	undef
Delay_on_driver_Is	s	Delay of the on signal low to high	undef	583.7p	1.376n	undef
Delay_off_driver_Is	s	Delay of the off signal high to low	undef	797.2p	1.878n	undef
Curr_cons_vdd2v5	A	Current consumption from the 2.5V domain	undef	18.89u	43.1u	undef
Curr_cons_vdd1v5	A	Current consumption from the 1.5V domain	undef	6.03n	147.3n	undef
Curr_leak_0b	A	Current leakage from 2.5V domain for input low	undef	158.8p	3.492u	undef
Curr_leak_1b	A	Current leakage from 2.5V domain for input high	undef	167.2p	3.489u	undef

Figure 47: Main measurements – montecarlo conditions

The target of this thesis is to provide a generator of Driver with good time performances, while maintaining limited consumption and occupied area. In particular, the specifications in typical condition are presented in Table 4.1.

Measure	Unit	Specification
Rise lson	s	< 1.5*n*
Fall lson	s	< 500*p*
Delay on driver _ls	s	< 1.5*n*
Delay off driver ls	s	< 1.2*n*
Curr cons vdd2v5	A	< 35*u*
Curr cons vdd1v5	A	< 10*n*
Curr leak 0B	A	< 5*n*
Curr leak 1B	A	< 5*n*

Table 4.1: Specifications in typical conditions

By comparing the measurements obtained with the specifications, it can be seen that the performances are met; in particular, the timing performances of interest are largely satisfied under nominal conditions, while under corner conditions the measurements remain close to acceptable values. On the other hand, consumption under nominal conditions remains limited to set values, while under corner conditions there is a wide variation, as it was expected.

4.2 WiCked Optimisation

The WiCked optimisation tool, presented in chapter 1, has been successfully used obtaining the complete workflow for the schematic generator. A python code has been written to exploit all the advantages of WiCked. Specifically, the WiCked gangway is made available for the user to import into the code and which allows connection to the tool. WiCked gangway enables starting the tool from Python for sizing, verification and optimization. The code is presented in Listing 4.2 and a description of how it is structured follows.

```
{ import wickedgangway

# input data directory

basedir=os.getcwd()

#####################################################

def driver(mode):

#####################################################

    'Sizing flow circuit driver'

    #wickedbatch class instance

    run = wickedgangway.standalone(projectname='driver')

    # print the relevant configuration

# run.print_configuration()

    # netlist(s)

    run.add_netlist(

#       netlistpath=os.path.join(basedir,'tb_driver.rs.tit'),
        netlistpath='/home/pad_ip_9/BAG/nodm/default/units/main/home/
        demirid/simulation/tb_driver/titan/schematic/netlist/tb_driver.

        rs.tit', simulator='titan',
              testbench_id='tb_driver.rs'

#    parameterization

run.add_parameterization(
deviceinstance='BOOST_DRIVER_LS_TAPERED_BUFFER,X/MN1_1',

properties=[ ('W', 'WN_MIN') ] )
```

```
 run.add_parameterization(
deviceinstance='BOOST_DRIVER_LS_TAPERED_BUFFER,X/MN2_2',

properties = [ ('W', 'WN_MIN*ALFA') ] )

 run.add_parameterization(
deviceinstance='BOOST_DRIVER_LS_TAPERED_BUFFER,X/MN3_3',

 properties = [ ('NF', 'ALFA'), ('W', 'WN_MIN*ALFA*ALFA') ] )

run.add_parameterization(

deviceinstance='BOOST_DRIVER_LS_TAPERED_BUFFER,X/MN4_4',

properties = [ ('M', 'ALFA'), ('NF', 'ALFA'), ('W', 'WN_MIN*ALFA* ALFA') ] )

 run.add_parameterization(
deviceinstance='BOOST_DRIVER_LS_TAPERED_BUFFER,X/MP1_1',

properties = [ ('W', 'WP_MIN') ] )

run.add_parameterization(
deviceinstance='BOOST_DRIVER_LS_TAPERED_BUFFER,X/MP2_2',

properties = [ ('W', 'WP_MIN*ALFA') ] )

run.add_parameterization(
deviceinstance='BOOST_DRIVER_LS_TAPERED_BUFFER,X/MP31_3',

 properties = [ ('NF', 'ALFA'), ('W', 'WP_MIN*ALFA*ALFA') ] )

 run.add_parameterization(
deviceinstance='BOOST_DRIVER_LS_TAPERED_BUFFER,X/MP32_3',

properties=[ ('NF', 'ALFA'), ('W', 'WP_MIN*ALFA*ALFA') ]

      )
```

```
run.add_parameterization( deviceinstance='BOOST_DRIVER_LS_TAPERED_BUFFER,X/MP4',
    properties=[ ('W', 'W_SWITCH') ] )

run.add_parameterization(deviceinstance='BOOST_DRIVER_LS_TAPERED_BUFFER,X/MP41_4',
    properties=[ ('M', 'ALFA'), ('NF', 'ALFA'), ('W', 'WP_MIN*ALFA* ALFA') ])

run.add_parameterization(deviceinstance='BOOST_DRIVER_LS_TAPERED_BUFFER,X/MP42_4',
    properties=[ ('M', 'ALFA'), ('NF', 'ALFA'), ('W', 'WP_MIN*ALFA* ALFA') ] )


run.add_parameterization(deviceinstance='BOOST_DRIVER_LS_TAPERED_BUFFER,
X/MP9', properties=[ ('W', 'W_SWITCH*ALFA') ] )

# Design parameters

run.add_design_parameter( parametername='ALFA', initial=4, lower=2, upper=7, grid
  =[(2,1,7)]

)

run.add_design_parameter( parametername='W_SWITCH', initial='2u', lower='0.5u', upper='15 u',
  grid = 'ifxdevsymbol c_*trans finger width *ana'

)

run.add_design_parameter( parametername='WN_MIN', initial='0.92u', lower='0.5u', upper=' 10u',
  grid = 'ifxdevsymbol c_*trans finger width *ana'

)

run.add_design_parameter(  parametername='WP_MIN', initial='2.84u', lower='0.5u', upper='

  10u', grid = 'ifxdevsymbol c_*trans finger width *ana'

)

# operating parameters run.add_operating_parameter( parametername='TEMPERATURE', initial='27',
lower='-40', upper= '175' ) run.add_operating_parameter( parametername='VDD2V5', initial='2.5',
lower='2.25', upper=' 2.75'

)
```

```
# Process corner

run.set_corner_mode('ce')  run.add_ce_corner('CORNER',
models=[

    ('/home/pad_ip_9/BAG/nodm/default/resources/.TECH/titan/models/ include.tit', ('FAST', 'NOM',
  'SLG')),

    ('/home/pad_ip_9/BAG/nodm/default/resources/.TECH/titan/models/ include_SOAC.tit',
  ('WARNINGS_OFF',)),

    ('/home/pad_ip_9/BAG/nodm/default/resources/.TECH/titan/models/ aging/AgeLib.tit', ('OFF',))

],)

# Sizing flow sizingsettings = {

    'twosteps':False,

    'constraints':wickedgangway.WORSTCASE,

    'wcoperationalgorithm':[wickedgangway.COORDINATESEARCH,
  wickedgangway.ALLCOMBINATIONS],

    'wcoperationaccuracy':0.05,

    'combinePoCvalues':False,

'displayconstraints':True,

    'usemanufacturinggrid':True,

    'feasalgorithm':wickedgangway.FINDCENTRAL,

    'maxiterations_feasibility':15,

    'maxiterations_nominal':6,

    'minmaxcostfunction':wickedgangway.EXPONENTIAL,

    'sensatworstcase':True,
```

```
} flow = wickedgangway.flow_wicked_optimization(run,'runNominal', sizingsettings,mode!='setup')

# specifications flow.add_area_specification('AREA_DRIVER', [

        '/tb_driver.rs/XI_DRIVER_LS/XI_TAPERED_BUFFER_0<0>/MN11',

        '/tb_driver.rs/XI_DRIVER_LS/XI_TAPERED_BUFFER_0<0>/MN1_1',

        '/tb_driver.rs/XI_DRIVER_LS/XI_TAPERED_BUFFER_0<0>/MN2_2',

        '/tb_driver.rs/XI_DRIVER_LS/XI_TAPERED_BUFFER_0<0>/MN3_3',

        '/tb_driver.rs/XI_DRIVER_LS/XI_TAPERED_BUFFER_0<0>/MN4_4',

        '/tb_driver.rs/XI_DRIVER_LS/XI_TAPERED_BUFFER_0<0>/MP1_1',

        '/tb_driver.rs/XI_DRIVER_LS/XI_TAPERED_BUFFER_0<0>/MP2_2',

        '/tb_driver.rs/XI_DRIVER_LS/XI_TAPERED_BUFFER_0<0>/MP4',

        '/tb_driver.rs/XI_DRIVER_LS/XI_TAPERED_BUFFER_0<0>/MP9',

        '/tb_driver.rs/XI_DRIVER_LS/XI_TAPERED_BUFFER_0<0>/MP31_3',

        '/tb_driver.rs/XI_DRIVER_LS/XI_TAPERED_BUFFER_0<0>/MP32_3',

        '/tb_driver.rs/XI_DRIVER_LS/XI_TAPERED_BUFFER_0<0>/MP41_4',

        '/tb_driver.rs/XI_DRIVER_LS/XI_TAPERED_BUFFER_0<0>/MP42_4',

        '/tb_driver.rs/XI_DRIVER_LS/XI_TAPERED_BUFFER_0<1>/MN11',

        '/tb_driver.rs/XI_DRIVER_LS/XI_TAPERED_BUFFER_0<1>/MN1_1',

        '/tb_driver.rs/XI_DRIVER_LS/XI_TAPERED_BUFFER_0<1>/MN2_2',

        '/tb_driver.rs/XI_DRIVER_LS/XI_TAPERED_BUFFER_0<1>/MN3_3',

        '/tb_driver.rs/XI_DRIVER_LS/XI_TAPERED_BUFFER_0<1>/MN4_4',

        '/tb_driver.rs/XI_DRIVER_LS/XI_TAPERED_BUFFER_0<1>/MP1_1',

        '/tb_driver.rs/XI_DRIVER_LS/XI_TAPERED_BUFFER_0<1>/MP2_2',

        '/tb_driver.rs/XI_DRIVER_LS/XI_TAPERED_BUFFER_0<1>/MP4',
```

```
’/tb_driver.rs/XI_DRIVER_LS/XI_TAPERED_BUFFER_0<1>/MP9’,
’/tb_driver.rs/XI_DRIVER_LS/XI_TAPERED_BUFFER_0<1>/MP31_3’,
’/tb_driver.rs/XI_DRIVER_LS/XI_TAPERED_BUFFER_0<1>/MP41_4’,
’/tb_driver.rs/XI_DRIVER_LS/XI_TAPERED_BUFFER_0<1>/MP42_4’,
’/tb_driver.rs/XI_DRIVER_LS/XI_TAPERED_BUFFER_0<2>/MN11’,
’/tb_driver.rs/XI_DRIVER_LS/XI_TAPERED_BUFFER_0<2>/MN1_1’,
’/tb_driver.rs/XI_DRIVER_LS/XI_TAPERED_BUFFER_0<2>/MN2_2’,
’/tb_driver.rs/XI_DRIVER_LS/XI_TAPERED_BUFFER_0<2>/MN3_3’,
’/tb_driver.rs/XI_DRIVER_LS/XI_TAPERED_BUFFER_0<2>/MN4_4’,
’/tb_driver.rs/XI_DRIVER_LS/XI_TAPERED_BUFFER_0<2>/MP1_1’,
’/tb_driver.rs/XI_DRIVER_LS/XI_TAPERED_BUFFER_0<2>/MP2_2’,
’/tb_driver.rs/XI_DRIVER_LS/XI_TAPERED_BUFFER_0<2>/MP4’,
’/tb_driver.rs/XI_DRIVER_LS/XI_TAPERED_BUFFER_0<2>/MP9’,
’/tb_driver.rs/XI_DRIVER_LS/XI_TAPERED_BUFFER_0<2>/MP31_3’,
’/tb_driver.rs/XI_DRIVER_LS/XI_TAPERED_BUFFER_0<2>/MP32_3’,
’/tb_driver.rs/XI_DRIVER_LS/XI_TAPERED_BUFFER_0<2>/MP41_4’,
’/tb_driver.rs/XI_DRIVER_LS/XI_TAPERED_BUFFER_0<2>/MP42_4’,
’/tb_driver.rs/XI_DRIVER_LS/XI_TAPERED_BUFFER_0<3>/MN11’,
’/tb_driver.rs/XI_DRIVER_LS/XI_TAPERED_BUFFER_0<3>/MN1_1’,
’/tb_driver.rs/XI_DRIVER_LS/XI_TAPERED_BUFFER_0<3>/MN2_2’,
’/tb_driver.rs/XI_DRIVER_LS/XI_TAPERED_BUFFER_0<3>/MN3_3’,
’/tb_driver.rs/XI_DRIVER_LS/XI_TAPERED_BUFFER_0<3>/MN4_4’,
’/tb_driver.rs/XI_DRIVER_LS/XI_TAPERED_BUFFER_0<3>/MP1_1’,
’/tb_driver.rs/XI_DRIVER_LS/XI_TAPERED_BUFFER_0<3>/MP2_2’,
```

```
        '/tb_driver.rs/XI_DRIVER_LS/XI_TAPERED_BUFFER_0<3>/MP4',
        '/tb_driver.rs/XI_DRIVER_LS/XI_TAPERED_BUFFER_0<3>/MP9',
        '/tb_driver.rs/XI_DRIVER_LS/XI_TAPERED_BUFFER_0<3>/MP31_3',
        '/tb_driver.rs/XI_DRIVER_LS/XI_TAPERED_BUFFER_0<3>/MP32_3',
        '/tb_driver.rs/XI_DRIVER_LS/XI_TAPERED_BUFFER_0<3>/MP41_4',
        '/tb_driver.rs/XI_DRIVER_LS/XI_TAPERED_BUFFER_0<3>/MP42_4',

], nominaltype_initial=wickedgangway.MIN, scaling='100p')
flow.add_specification('RISE_LSON', analysis='all',
upper_initial=f'{713.7*1.1}p', nominaltype_initial=wickedgangway
.BOUND, upper_worstcase='1.2n', nominaltype_worstcase=
    wickedgangway.BOUND) #tran flow.add_specification('FALL_LSON', analysis='all',
  upper_initial=f'{388.9*1.1}p', nominaltype_initial=wickedgangway
    .BOUND, upper_worstcase='880p', nominaltype_worstcase= wickedgangway.BOUND) #tran
  flow.add_specification('DELAY_ON_DRIVER_LS', analysis='all',
   upper_initial=f'{837.2*1.1}p', nominaltype_initial=wickedgangway
    .BOUND, upper_worstcase='1.8n', nominaltype_worstcase = wickedgangway.BOUND) #tran
  flow.add_specification('DELAY_OFF_DRIVER_LS', analysis='all',
    upper_initial=f'{1.167*1.1}n', nominaltype_initial=wickedgangway
    .BOUND, upper_worstcase='2n', nominaltype_worstcase= wickedgangway.BOUND) #tran
  flow.add_specification('CURR_CONS_VDD2V5', analysis='all', scaling='100n',
  nominaltype_initial=wickedgangway.MIN ) #tran
```

```
    flow.add_specification('CURR_LEAK_0B',          analysis='all',          scaling='100p',
    nominaltype_initial=wickedgangway.MIN ) #dc

    flow.add_specification('CURR_LEAK_1B',          analysis='all',          scaling='100p',
    nominaltype_initial=wickedgangway.MIN)#dc
    flow.add_specification('CURR_CONS_VDD1V5', analysis='---' )

    # print setup data

    print(run) print(flow)



    # run WiCkeD optimization run.wicked( flow=flow, mode=mode, simulators=4, #should not be
    larger than the number of used

      cores

    )

}
```

Listing 4.2: driver function for optimization

The starting point is to specified for the code the runset file, its path and the simulator. The runset file for the driver contains a full set of informations for the program to be run (simulation), plus the locations (paths) of all necessary (input & output) files. rs; specifically it has the informations of outputs to be saved, corners, analysis to be done, variable definitions and the pointing to the *nd* file which contains the actual netlist, i.e. the status of the circuit. The runset file will be the reference for the code, together with the simulator to be used. At this point follows the WiCked state of the circuit, that is the schematic parametrization according to the design criterion.

Four design parameters has been defined:

- alfa: the tapered factor (the index of increase) for the driver
- wn min: minimum nmos width of the first stage
- wp min: minimum pmos width of the first stage
- w _switch: minimum width for the first switch in the third stage

The defined parameters can be controlled through the gangway, perform the optimization and find the optimum values to reach specific performances. The parameterization of the circuit is done with the four parameters to implement the criterion illustrated in Chapter 2. For each of these parameters, an initial value and a range with a minimum and maximum value has been specified.

Operating parameters, which are temperature and power supply, and process corners are also defined. These are necessary for the circuit because it is tested in all combinations of these parameters to assess how performances vary depending on operating conditions. This first part of the code constitutes the constraint editor, that is, the initial setup of the tool. Then follows the sizing flow in which the type of optimization and measures to be optimized are defined. There are three types of optimisation: nominal, yeld and feasibility. For the code the nominal optimization has been chosen.

The measures of interest are presented in the previous section; an other measure has been added to the flow:

- Area: is the measure that takes into account the area occupied by the driver,

  i.e. the devices that are subject to code sizing

For each of these measurements, the initial value was specified and how to optimize them. In particular, for time measurements has been given an upper bound under nominal condition and worst-case condition to not be exceeded; no lower bound has been specified, because for this type of evaluations it is of interest to achieve the shortest possible time.

Area and current measurements were specified to minimise as much as possible and an equivalent improvement index between the various measurements called *scaling* was indicated (e.g. a decrease of 100n for the Curr cons vdd2v5 is equivalent to a decrease of 100p for Area occupied).

The Tables 4.2 can be consulted for greater clarity of the conditions imposed on the optimizer.

Measure	Lower Bound	Initial Value	Nominal Upper Bound
Rise lson	/	713.7p	713.7p +10%
Rise lsoff	/	388.9p	388.9p + 10%
Delay on driver ls	/	837.2p	837.2p + 10%
Delay off driver ls	/	1.167n	1.167n + 10%
Curr cons vdd2v5	/	27.87u	/
Curr leak 0B	/	2.238n	/
Curr leak 1B	/	2.985n	/
Area	/	MIN	MIN

Measure	Worst Case Upper Bound	Scaling
Rise lson	1.2n	/
Rise _lsoff	880p	/
Delay _on driver ls	1.8n	/
Delay off _driver ls	2n	/
Curr cons vdd2v5	/	100n
Curr leak 0B	/	100p
Curr leak 1B	/	100p
Area	/	100p

Table 4.2: Tables for measurements specifications

It is important to note that under nominal conditions the upper bound is much more stringent than the specification required for the driver, in fact a variation of only 10% of the simulation value was considered. The same applies to the worst case where strict conditions have been imposed. Furthermore, measurement Curr cons vdd1v5 was not taken into account as the Driver is a device supplied with 2.5V.

Finally, the code can be launched. The result will provide the optimal values of the defined parameters in order to fulfil the conditions imposed on the measurements. For a detailed study of the result, the tool can be opened in GUI mode and observe how the optimiser performed through tables and graphs.

4.2.1 Wicked Flow

The use of the tool was included in the schematic generator in order to have the complete workflow. In the code, it is proposed to perform the optimization with WiCked after the

trade-off sizing is carried out. Listing 4.3 presents the part of the code added within the schematic generator to offer the possibility of using the optimizer.

```
User = input("\u001b[36m[INFO]\u001b [0m "+"

Do you want to launch wicked? \u001b[32m(y/n)\u001b[0m: ")

 sizing = None

if user == 'y':

      user = input("\u001b[36m[INFO]\u001b[0m "+ "Mode: batch (b)/

      GUI (g)/ setup (s) \u001b[32m(b/g/s)\u001b[0m: ")

       if user == 'b': sizing = driver.driver('batch')

       elif user == 'g': sizing = driver.driver('gui')

       elif user == 's': driver.driver('setup')

      if sizing:

                print('Final sizing:')

                for p,v in sizing.items(): print(f' {p} = {driver.wickedgangway.to_eng(v)}')

      driver_wicked_sizing(self, sizing)

       print(interspace)

      print("\n\u001b[36m[INFO]\u001b[0m " + "Mosfet dimension after WICKED
      optimization:\n") print(get_string_mos_ti_parameters(self))

      print(interspace)

      else:

            print('No final sizing. Refer to above.')
```

Listing 4.3: Code for user interaction with WiCked

In order for the driver to be automatically resized with the optimal values provided by the tool, a method has been written and is presented in Listing 4.4.

```
# Method to size the driver according to optimum values of wicked

def driver_wicked_sizing(self, sizing):

    # Sizing done in this method follows the same parametrization done in WICKED
    tapered_factor = sizing["ALFA"]

    # Sizing 1 stage with the new optimum value

    self.sch_gen.set_ti_parameter("N1_1", "w", sizing["WN_MIN"])

    self.sch_gen.set_ti_parameter("P1_1", "w", sizing["WP_MIN"])

    # Sizing 2 stage

    self.sch_gen.set_ti_parameter("N2_2", "w", float(self.sch_gen.
    instances["N1_1"]["parameters"]["w"]["value"]) * round( tapered_factor))
    self.sch_gen.set_ti_parameter("P2_2", "w", float(self.sch_gen.

    instances["P1_1"]["parameters"]["w"]["value"]) * round( tapered_factor))

    # Sizing 3 stage

    # Nmos stage

    self.sch_gen.set_ti_parameter("N3_3", "w", float(self.sch_gen.

    instances["N2_2"]["parameters"]["w"]["value"])* round( tapered_factor))
    self.sch_gen.set_ti_parameter("N3_3", "nf", round( tapered_factor))

    # Pmos stage

    self.sch_gen.set_ti_parameter("P31_3", "w", float(self.sch_gen.
```

```
        instances["P2_2"]["parameters"]["w"]["value"])*round( tapered_factor))
        self.sch_gen.set_ti_parameter("P31_3", "nf", round( tapered_factor))
        self.sch_gen.set_ti_parameter("P32_3", "w", float(self.sch_gen.

        instances["P2_2"]["parameters"]["w"]["value"])*round( tapered_factor))
self.sch_gen.set_ti_parameter("P32_3", "nf", round( tapered_factor))

 # Sizing 4 stage

 # Nmos stage

self.sch_gen.set_ti_parameter("N4_4", "w",  float(self.

sch_gen.instances["N3_3"]["parameters"]["w"]["value"]))   self.sch_gen.set_ti_parameter("N4_4",
"multi", round( tapered_factor))

self.sch_gen.set_ti_parameter("N4_4","nf",round( tapered_factor))

# Pmos stage

self.sch_gen.set_ti_parameter("P41_4", "w", float(self.

sch_gen.instances["P31_3"]["parameters"]["w"]["value"]))
self.sch_gen.set_ti_parameter("P41_4","nf",round(tapered_factor))
self.sch_gen.set_ti_parameter("P41_4","multi",round(tapered_factor))
self.sch_gen.set_ti_parameter("P42_4", "w", float(self.

sch_gen.instances["P32_3"]["parameters"]["w"]["value"]))
self.sch_gen.set_ti_parameter("P42_4","nf",round(tapered_factor))
self.sch_gen.set_ti_parameter("P42_4", "multi", round( tapered_factor))

 # Sizing switch P4 and P9 of the third and fourth stage

respectively

self.sch_gen.set_ti_parameter("P4", "w", sizing["W_SWITCH"])
```

```
self.sch_gen.set_ti_parameter("P9", "w", float(self.sch_gen.

instances["P4"]["parameters"]["w"]["value"]) * round( tapered_factor))

return None
```

Listing 4.4: Method that sizes the driver according WiCked values

The user interaction flow offers the possibility of launching the tool in three modes: batch, GUI and setup mode. Batch mode is chosen to have a fully automatic flow. The option GUI allows to open the interface in order to analyse the results with graphs and tables. Of particular interest is the worst-case analysis in Figure 48.

Summary | Details

Performance or Constraint	Bound Type	Progress	Specification	Nominal Value	Worst-Case Value	Diff.	Diff. / Nominal	Worst-Case To Spec Margin	Rel. Margin	TEM...	VDD2V5	include.tit
DELAY_ON_DRIV...	Lower	finished	min	838.8 p	545.8 p	-293 p	-35%			-40	2.75	FAST
	Upper	finished	< 1.8 n	838.8 p	1.542 n	+703.2 p	84%	258 p		175	2.25	SLG
FALL_LSON	Lower	finished	min	369.6 p	188.8 p	-180.7 p	-49%			-40	2.75	FAST
	Upper	finished	< 880 p	369.6 p	822.4 p	+452.8 p	> 100%	57.6 p		175	2.25	SLG
DELAY_OFF_DRI...	Lower	finished	min	1.05 n	636.7 p	-412.8 p	-39%			-40	2.75	FAST
	Upper	finished	< 2 n	1.05 n	1.982 n	+932.2 p	89%	18.28 p		175	2.25	SLG
RISE_LSON	Lower	finished	min	564.3 p	237.8 p	-326.6 p	-58%			-40	2.25	FAST
	Upper	finished	< 1.2 n	564.3 p	1.2 n	+635.2 p	> 100%	449.7 f		175	2.66	SLG

Figure 48: Worst case table

The tool builds a table with various options that can be referred to in order to evaluate the results. In particular, from Figure 48, given the four time measures, it can be seen the specification for the measurements. At this point, the Nominal column offers the simulation results in nominal conditions, while the Worst-case value column offers the values of lower and upper extremes that a measurement can assume. Specifically, in the last three columns there are the operating conditions that cause the circuit to behave in that particular way. Finally, there are columns that indicate the deviation from the nominal value both in absolute and relative percentages, with an associated margin from the specified maximum value.

In addition to this interesting table, there are graphs of how measures change iteration after iteration. In Figure 49 and 50 timing performances can be observe in both the nominal (green) and worst case (blue) condition. In all four measures, the upper bound is respected.

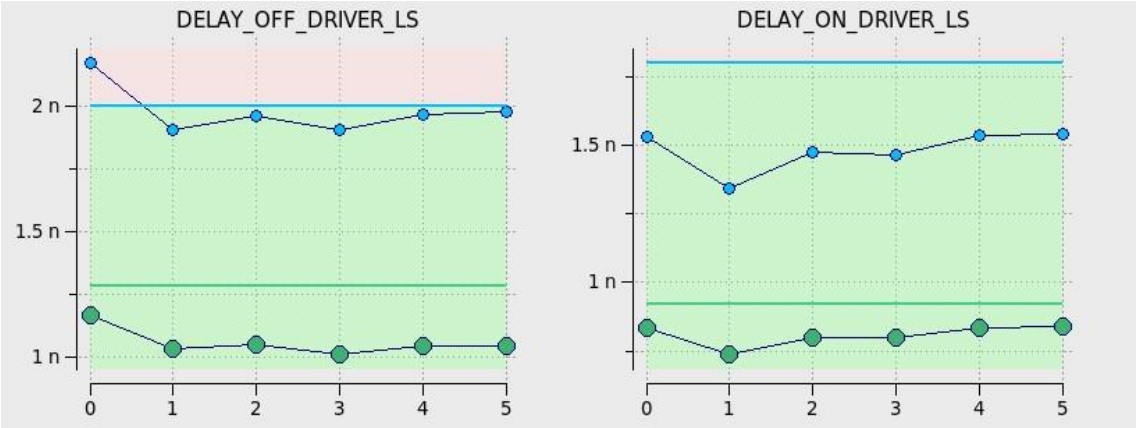

Figure 49: Delay on and off measures in WiCked

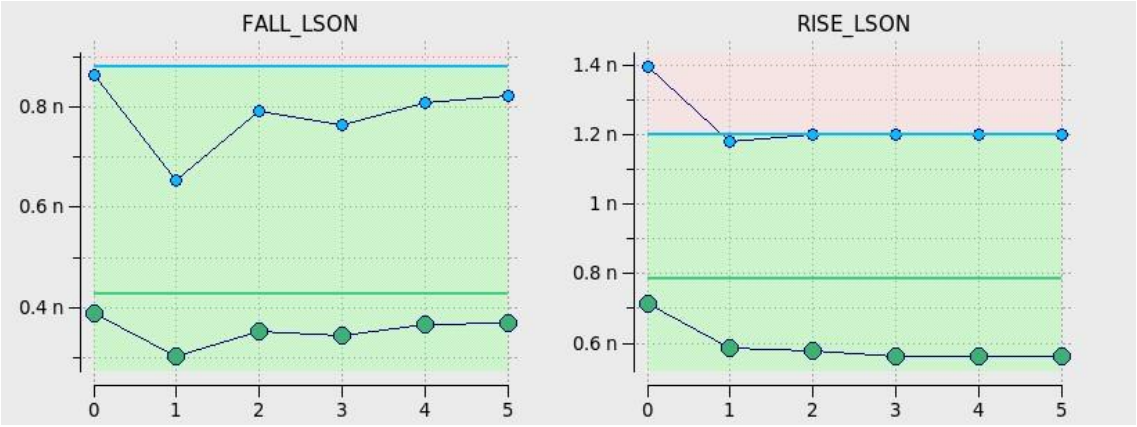

Figure 50: Fall and rise measures in WiCked

Regarding the remaining current and area measures, having not specified a bound, but simply imposed to minimise them, WiCked's algorithm works in such a way that it first satisfies the measurements with a bound, then in successive iterations minimises the remaining measurements. This explains the initial oscillating pattern. Figure 51 and 52 present the results.

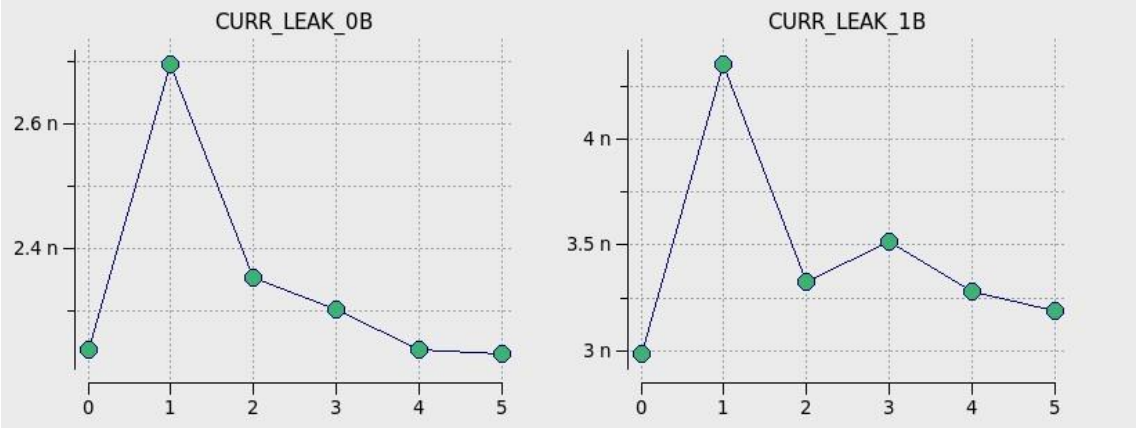

Figure 51: Current leakage measure in WiCked

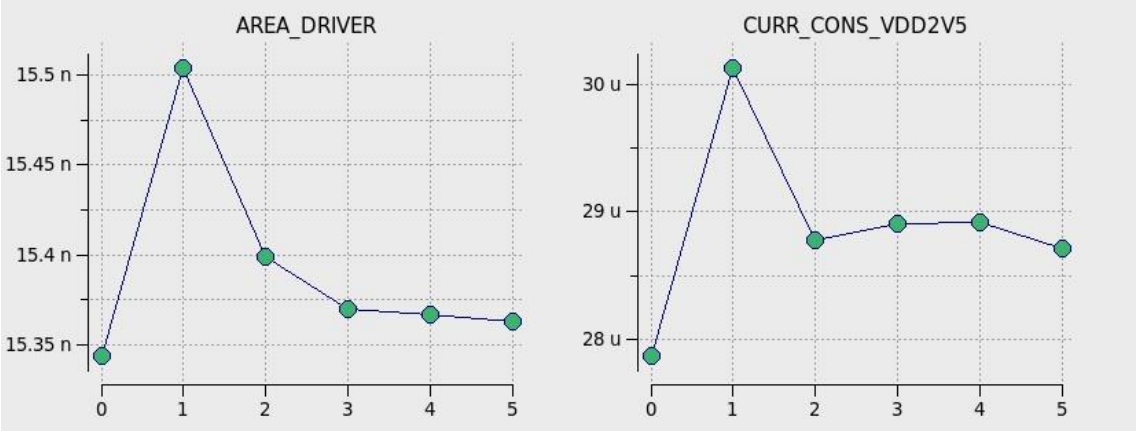

Figure 52: Area and current dynamic measure in WiCked

In Figure 53 there is the legend for the graphs.

Figure 53: Legend for the WiCked graphs

Finally, we have the values for the parameters defined earlier to be used to achieve optimum performance. The code prints out a summary table on the screen with the various measurements and specifying whether or not the test conditions have passed, followed by the final dimensioning values, as in Figure 54.

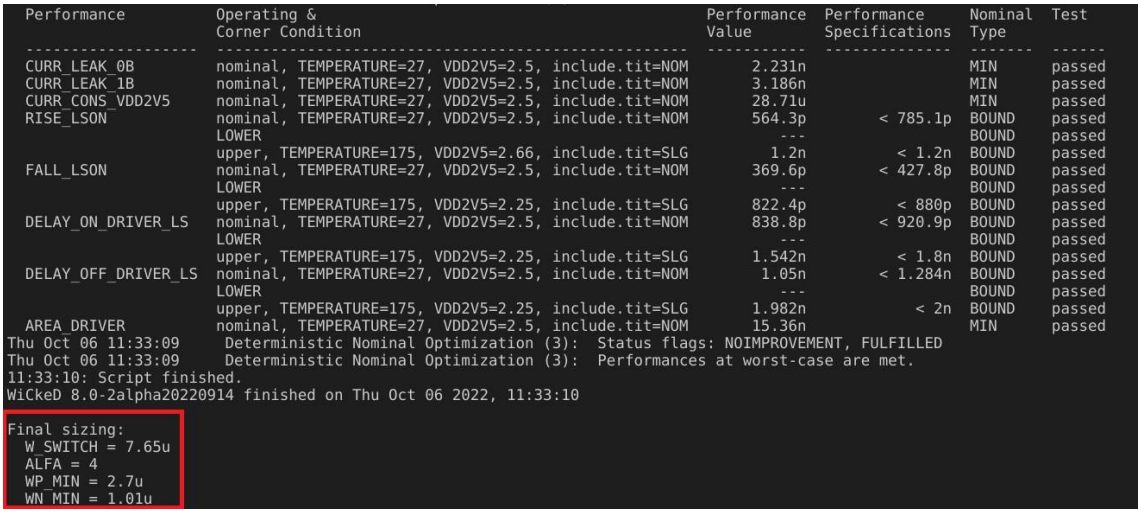

Figure 54: WiCked table summary

The code automatically resizes the driver mosfets using the measurements given by WiCked and implementing the chosen criterion. It makes the performed update visible to the user as in Figure 55.

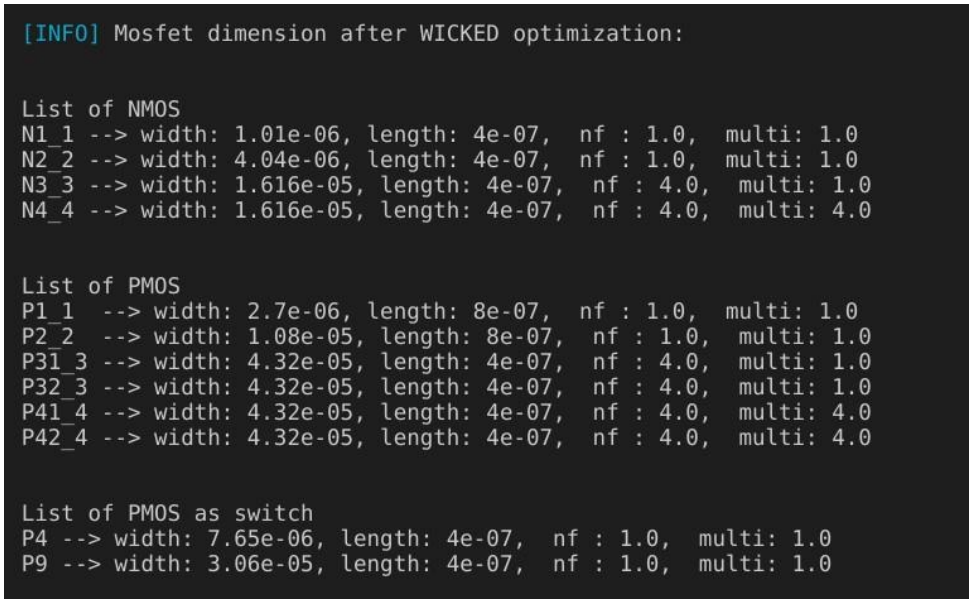

Figure 55: Final sizing

4.2.2 Verification Result

Three tests are presented as in the previous section.

TEST1

measure	unit	description	specMin	measMin	measMax	specMax
Rise_Ison	s	Ison rise time	undef	564.3p	564.3p	undef
Fall_Ison	s	Ison fall time	undef	369.6p	369.6p	undef
Delay_on_driver_Is	s	Delay of the on signal low to high	undef	838.8p	838.8p	undef
Delay_off_driver_Is	s	Delay of the off signal high to low	undef	1.05n	1.05n	undef
Curr_cons_vdd2v5	A	Current consumption from the 2.5V domain	undef	28.71u	28.71u	undef
Curr_cons_vdd1v5	A	Current consumption from the 1.5V domain	undef	8.386n	8.386n	undef
Curr_leak_0b	A	Current leakage from 2.5V domain for input low	undef	2.231n	2.231n	undef
Curr_leak_1b	A	Current leakage from 2.5V domain for input high	undef	3.186n	3.186n	undef

Figure 56: Main measurements with WiCked sizing – typical conditions

TEST2

measure	unit	description	specMin	measMin	measMax	specMax
Rise_Ison	s	Ison rise time	undef	225.3p	1.18n	undef
Fall_Ison	s	Ison fall time	undef	187.9p	839.2p	undef
Delay_on_driver_Is	s	Delay of the on signal low to high	undef	541.1p	1.573n	undef
Delay_off_driver_Is	s	Delay of the off signal high to low	undef	629.6p	2.024n	undef
Curr_cons_vdd2v5	A	Current consumption from the 2.5V domain	undef	18.31u	44.23u	undef
Curr_cons_vdd1v5	A	Current consumption from the 1.5V domain	undef	6.281n	219.2n	undef
Curr_leak_0b	A	Current leakage from 2.5V domain for input low	undef	158.2p	4.751u	undef
Curr_leak_1b	A	Current leakage from 2.5V domain for input high	undef	165.2p	6.525u	undef

Figure 57: Main measurements with WiCked sizing – corner conditions

TEST3

measure	unit	description	specMin	measMin	measMax	specMax
Rise_lson	s	lson rise time	undef	238.4p	947.2p	undef
Fall_lson	s	lson fall time	undef	223.8p	682.2p	undef
Delay_on_driver_ls	s	Delay of the on signal low to high	undef	592.8p	1.402n	undef
Delay_off_driver_ls	s	Delay of the off signal high to low	undef	728.1p	1.741n	undef
Curr_cons_vdd2v5	A	Current consumption from the 2.5V domain	undef	19.18u	44.52u	undef
Curr_cons_vdd1v5	A	Current consumption from the 1.5V domain	undef	6.03n	147.3n	undef
Curr_leak_0b	A	Current leakage from 2.5V domain for input low	undef	159.1p	3.425u	undef
Curr_leak_1b	A	Current leakage from 2.5V domain for input high	undef	167.8p	3.704u	undef

Figure 58: Main measurements with WiCked sizing – Montecarlo conditions

Sizing with WiCked values provides improved performances, especially in timing performances there is a clear improvement. Table 4.2.2 illustrates the relative percentage improvement to show the result achieved with WiCked.

Measure	Percentage improvement
Rise lson	21%
Rise lsoff	5%
Delay on _driver ls	/
Delay off driver _ls	10%

Table 4.3: Tables for timing performances improvement

The graphs shown in Figure 49 and 50 are also important to consider. In fact, if examined properly, it can be seen that even stricter constraints can be set for nominal conditions, leading to an even better solution. The currents consumption have maintained the same values, with some very small percentage deviations.

4.3 Layout generator: Driver

The workflow led to optimise the schematic, resulting in better electrical performance. At this point, the layout must also be updated with the new values. The layout generator developed allows the schematic layout to be instantiated with the new parameters, without the need to recreate everything from scratch. The coding of the layout generator offers these great advantages: flexibility and adaptability.

Figure X shows the new layout resulting DRC and LVS clean.

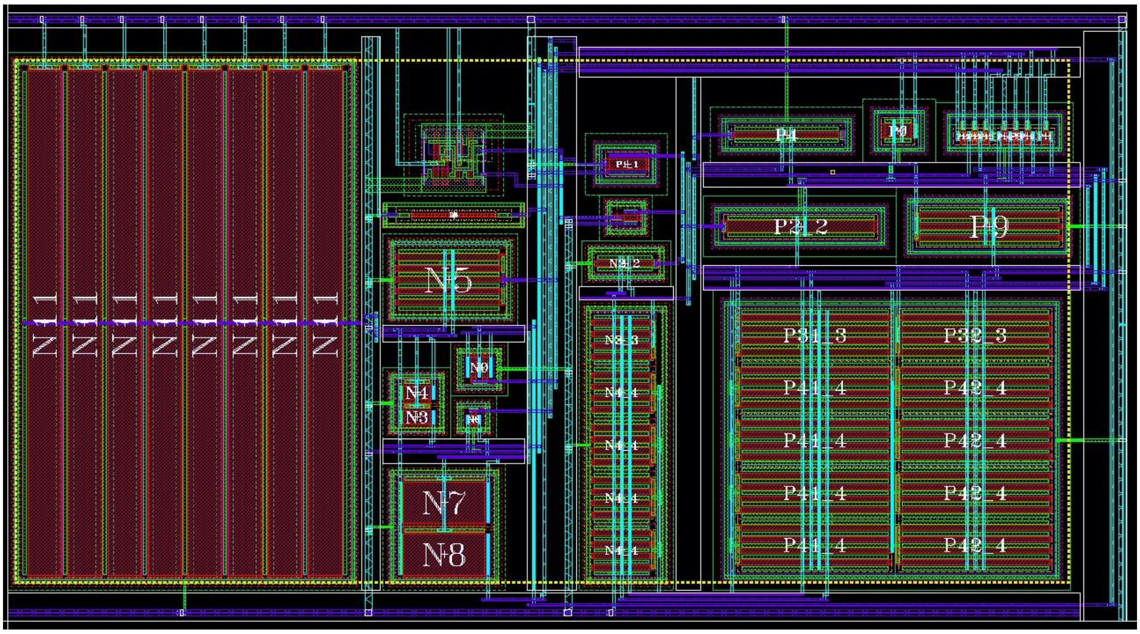

Figure 59: Final Layoutafter WiCked sizing

At this point, a post-layout verification is performed. The verification is done through simulations in Avenue, as done for the schematic. In order to do post-layout simulation the layout extraction first is needed as an intermediate step. Layout extraction is the translation of the topological layout back into the electrical circuit it is intended to represent. In this way it can be simulated. In addition, parasitic effects, specifically resistances and capacitance, are also taken into account. The minimum tolerance thresholds set for the parasitic resistance and capacitance contributions are $1m\Omega$ and $1fF$ respectively.

4.3.1 Post-layout verification

The circuit is tested and the results for the three tests are given as follows.

TEST 1

measure	unit	description	specMin	measMin	measMax	specMax
Rise_Json	s	Json rise time	undef	981.6p	981.6p	undef
Fall_Json	s	Json fall time	undef	499.4p	499.4p	undef
Delay_on_driver_Js	s	Delay of the on signal low to high	undef	1.031n	1.031n	undef
Delay_off_driver_Js	s	Delay of the off signal high to low	undef	1.254n	1.254n	undef
Curr_cons_vdd2v5	A	Current consumption from the 2.5V domain	undef	29.32u	29.32u	undef
Curr_cons_vdd1v5	A	Current consumption from the 1.5V domain	undef	8.391n	8.391n	undef
Curr_leak_0b	A	Current leakage from 2.5V domain for input low	undef	2.276n	2.276n	undef
Curr_leak_1b	A	Current leakage from 2.5V domain for input high	undef	3.235n	3.235n	undef

Figure 60: Main measurements with WiCked sizing – Typical conditions – Post layout Verification

TEST 2

measure	unit	description	specMin	measMin	measMax	specMax
Rise_Json	s	Json rise time	undef	393.1p	1.955n	undef
Fall_Json	s	Json fall time	undef	262.7p	1.082n	undef
Delay_on_driver_Js	s	Delay of the on signal low to high	undef	666.6p	1.923n	undef
Delay_off_driver_Js	s	Delay of the off signal high to low	undef	752.1p	2.406n	undef
Curr_cons_vdd2v5	A	Current consumption from the 2.5V domain	undef	18.94u	44.68u	undef
Curr_cons_vdd1v5	A	Current consumption from the 1.5V domain	undef	6.282n	219.2n	undef
Curr_leak_0b	A	Current leakage from 2.5V domain for input low	undef	343.3p	4.483u	undef
Curr_leak_1b	A	Current leakage from 2.5V domain for input high	undef	381.4p	6.211u	undef

Figure 61: Main measurements with WiCked sizing – Corner conditions – Post Layout Verification

TEST 3

measure	unit	description	specMin	measMin	measMax	specMax
Rise_Ison	s	Ison rise time	undef	382.4p	1.618n	undef
Fall_Ison	s	Ison fall time	undef	303.7p	901.2p	undef
Delay_on_driver_ls	s	Delay of the on signal low to high	undef	729.8p	1.717n	undef
Delay_off_driver_ls	s	Delay of the off signal high to low	undef	867.9p	2.074n	undef
Curr_cons_vdd2v5	A	Current consumption from the 2.5V domain	undef	19.75u	44.82u	undef
Curr_cons_vdd1v5	A	Current consumption from the 1.5V domain	undef	6.042n	147.3n	undef
Curr_leak_0b	A	Current leakage from 2.5V domain for input low	undef	344.4p	3.357u	undef
Curr_leak_1b	A	Current leakage from 2.5V domain for input high	undef	383.8p	3.467u	undef

Figure 62: Main measurements with WiCked sizing – Montecarlo conditions – Post Layout Verification

As expected, the performance of the layout varies with respect to the verification simulation done directly on the schematic, due to all the effects taken into account. However, the actual performance is very convincing, despite the fact that the layout is taken into account. In fact, referring to Table 4.1, it can be seen that the specifications are all met. Particularly under nominal conditions, current consumption remains below specification, not varying much from the schematic verification simulations. Regarding time performance, only the delay off exceeds the specification by 0.054 ns, which is considered acceptable as a variance; the other timings meet all the set specifications.

CHAPTER 5

CONCLUSION

The presented results show that the reuse of blocks through a coded methodology is not only possible but has been achieved. The sizing for the schematic generators matched the requested specifications under typical conditions, which were the target of the thesis, but also in some cases even the specifications at corner conditions are met. It can be stated that generators really speed up the design process and can be applied to the blocks which are commonly used in several applications but require different specifications. The design cycle is not completed, since optimization loops are needed such as the ones to also satisfy the given specifications in the corner conditions or to minimize even more the area consumption; this is something that will be investigated in future works. To have an even more compact layout and so a more competitive product, the layout tool needs to be updated and improved with the introduction of a more flexible routing algorithm. A more flexible generation of the basic modules could also increase the quality of the layout itself by allowing the user to fully express its expertise. The introduction of modules for the missing devices like capacitors, logic ports or even abstract modules is necessary in order to make Qgen a really efficient, complete and competitive layout tool.

REFERENCES

[1] H. G. a. R. S. J. Gerlach, "Analog Circuit Generator Design with a Flexible Hierarchical Framework", 2017.

[2] R. K. U. M. T. S. M. P. M. &. R. A. Frevert, High Sigma Yield Analysis & Optimization with WiCkeD, 2007.

[3] F. I. E. a. M. GmbH, Optimization of Mixer Circuits with WiCkeD, 2007.

s M. D. a. G. M. A. Balboni, Qgen: An Automatic Layout Generator for Analog Circuit Design, 2016.

[5] F. M. a. L. Malcher, Analog Layout Optimization Using Qgen Framework, 2020.

[6] S. L. a. M. S. S. H. Goh, Full-Custom and Semi-Custom Design for Low-Power Integrated Circuits, 2019.

[7] J. L. P. a. M. V. Amparo, Full-Custom and Semi-Custom Layout Generation Techniques, 2005.

[8] H. B. Kaushik, D. C. Rai and S. K. Jain, "Uniaxial compressive stress-strain model for clay brick masonry," *Current Science,* vol. 92, no. 4, pp. 497-501., 2007.