Dissertation

submitted to the

Combined Faculty of the Natural Sciences and Mathematics

of the

Ruprechts-Karls University
Heidelberg

for the degree of

Doctor of Natural Sciences

put forward by

M.Sc. Computer Engineering Tobias Markus

Born in
Lingen(Ems), Germany

March 25, 2021

Circuit Design Automation for High Speed Interconnects in Advanced Nodes

Advisor: Prof. Dr.-Ing. Ulrich Brüning

Date of oral examination: .....

Abstract

Design complexity has become is a rising issue in modern Mixed-Signal SOC designs. Especially today's Full-Custom flow lags automation. This thesis tackles this issue in introducing design flows and methods to create expert knowledge-based parametrizable schematics and layouts.

In this work, a seamless language bridge between python and SKILL was developed to interface with the commonly used design environment. The skillbridge was open-sourced and is widely used.

For the schematic a design flow is introduced which guarantees consistency between system-level and implementation. In sizing scripts expert knowledge can be encoded. These scripts are technology agnostic sizing scripts with high reusability.

For layout generation, reoccurring layout patterns were identified in industrial design projects and implemented as parametrizable elementary cells. In an initial version, these elementary cells were implemented with a commercial tool the Cadence PCell Designer. Based on lessons learned an own constraint-based layout generation framework was implemented, the XCell framework. The XCell framework offers interfaces to describe tool and technology agnostic generators.

With this framework, the elementary cells and many more generators were implemented and successfully used in larger designs. In leaf cells using elementary cells a huge part of the including shapes are generated. When these shapes were sorted by category it was shown that in the local category which includes the DRC critical layers 80% was generated.

With this, a significant reduction in design time with reusable layout cells was achieved. This effect is amplified because of all elementary cells being DRC clean and DFM friendly and thus reducing design turnaround cycles.

Zusammenfassung

Design Komplexität ist eines der aktuellsten Probleme im Design von Mixed-Signal SOCs. Der Full-Custom Designflow ist wenig automatisiert. Diese Arbeit führt einen Designflow und Methodiken ein, um Expertenwissen in parametrierbaren Schaltplan und Layout Generatoren zu implementieren.
Im Rahmen dieser Arbeit wurde mit der skillbridge ein nahtloses Interface zwischen Python und SKILL entwickelt. SKILL ist die Programmiersprache, mit welcher sich das meist genutzte Design Environment automatisieren lässt. Die skillbridge wurde als open-source Projekt veröffentlicht und ist mittlerweile viel genutzt.

Für die Schaltplanphase wurde ein Designflow eingeführt, welcher Konsistenz garantiert. In Dimensionierungsskripten kann Expertenwissen codiert werden. Diese Skripte sind Technology unabhängig und wiederverwendbar.
Für die Layoutphase wurden in einem industriellen Design Projekt sich wiederholende Layout Blöcke identifiziert und als parametrisierende Zellen implementiert. In einer initialen Version wurden diese als sogenannte Elementarzellen mit dem kommerziellen Tool Cadence PCell Designer implementiert. Basierend auf Resultaten wurde gelernt und ein eigenes Constraint basierendes Layout Generator Framework implementiert das XCell Framework. Das XCell Framework bietet ein Interface, um technologieunabhängige Generatoren zu beschreiben.

Mit diesem Framework wurden die Elementarzellen implementiert und viele weitere Generatoren. Alle Generatoren wurden erfolgreich in umfangreichen Layouts genutzt. In Leaf Zellen, in welchen diese Zellen benutzt wurden, wurde ein Großteil der Shapes des Layouts generiert. Wenn diese Shapes nach Kategorie sortiert werden, kann gezeigt werden das DRC kritische Layer zu 80% generierte Shapes enthält.
Somit ist eine signifikante Reduktion in der Designzeit mittels der wiederverwendbaren Layoutzellen möglich. Dieser Effekt wird weiter dadurch verstärkt, dass die Elementarzellen in ein DRC sauberes und DFM freundliches Layout resultieren und so Designzyklen reduzieren.

Acknowledgments

This thesis would not have been possible without the feedback and support of many people.

I would first like to thank Prof. Ulrich Brüning who made this work possible, gave insightful feedback and advice. Many thanks to my colleagues from the Computer Architecture Group and Extoll GmbH for the valuable discussions, patient advice, and constructive criticism.

Also, I would like to thank my friends, family, and Anne-Maria for always being there for me, their interest in my work, and the encouragement.

Contents

List of Abbreviation

SOC System on Chip

DRC Design Rule Check

DFM Design for Manufacturing

HPC High-Performance Computing

EM Electromigration

AMSRF Analog, Mixed-Signal and RF

SERDES Serializer Deserializer

NRE Non-Recurring Engineering

EDA Electronic Design Automation

RNM Real Number Model

HDL Hardware Description Language

AMS Analog-Mixed-Signal

LVS Layout Versus Schematic

SOI Silicon On Insulator

FDSOI Fully Depleted Silicon On Insulator

PDK Process Design Kit

CIW Command Interface Window

EMIR Electromigration/ IR drop

IPC Inter-Process Communication

DCO Digitally Controlled Oscillator

PLL Phase Lock Loop

1 Introduction

Complementary Metal Oxide Semiconductors (CMOS) have shown an unbelievable increase in transistor densities. The last decades resulted in smaller technology nodes and System on Chip (SoC) Designs with more and more transistors starting from the early 80s. It follows an exponential growth over the last decades. Moore's Law still stays and leads to complex designs yielding around 30 Billion Transistors. These complex designs are FPGAs (Versal VC1902, Ultrascale VU19P) and HPC processors like the AMD Epyc Rome seen in figure 1.2.

The demand for computing power in High-Performance Computing (HPC), AI and 5G

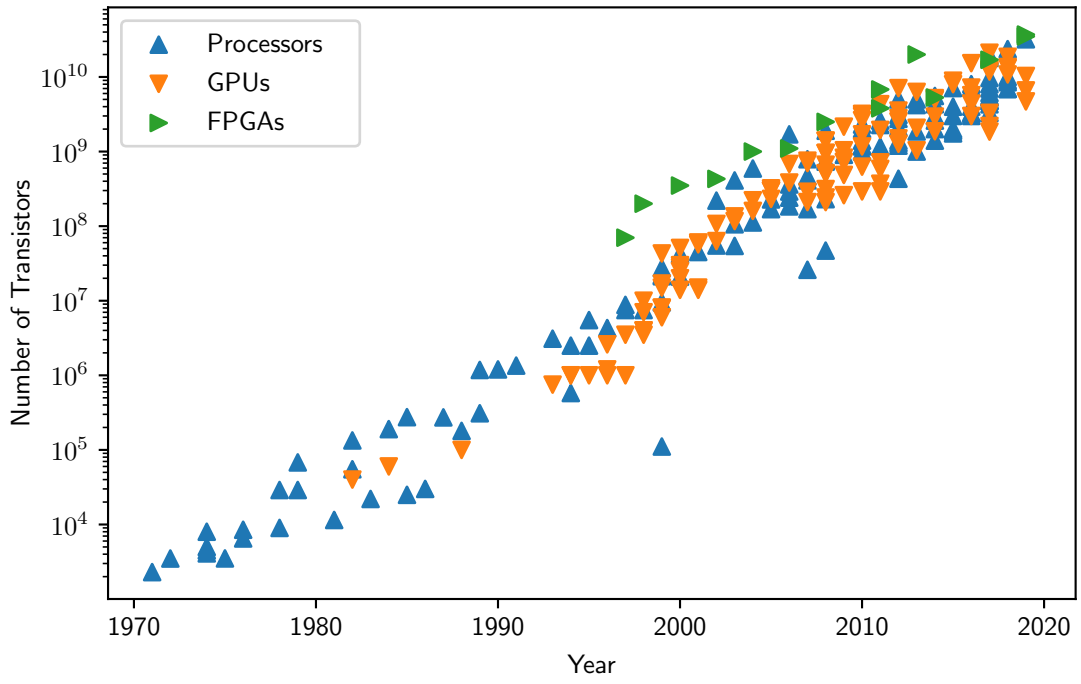


Figure 1.1: Transistor count in complex SOCs until 2020, data from [1]

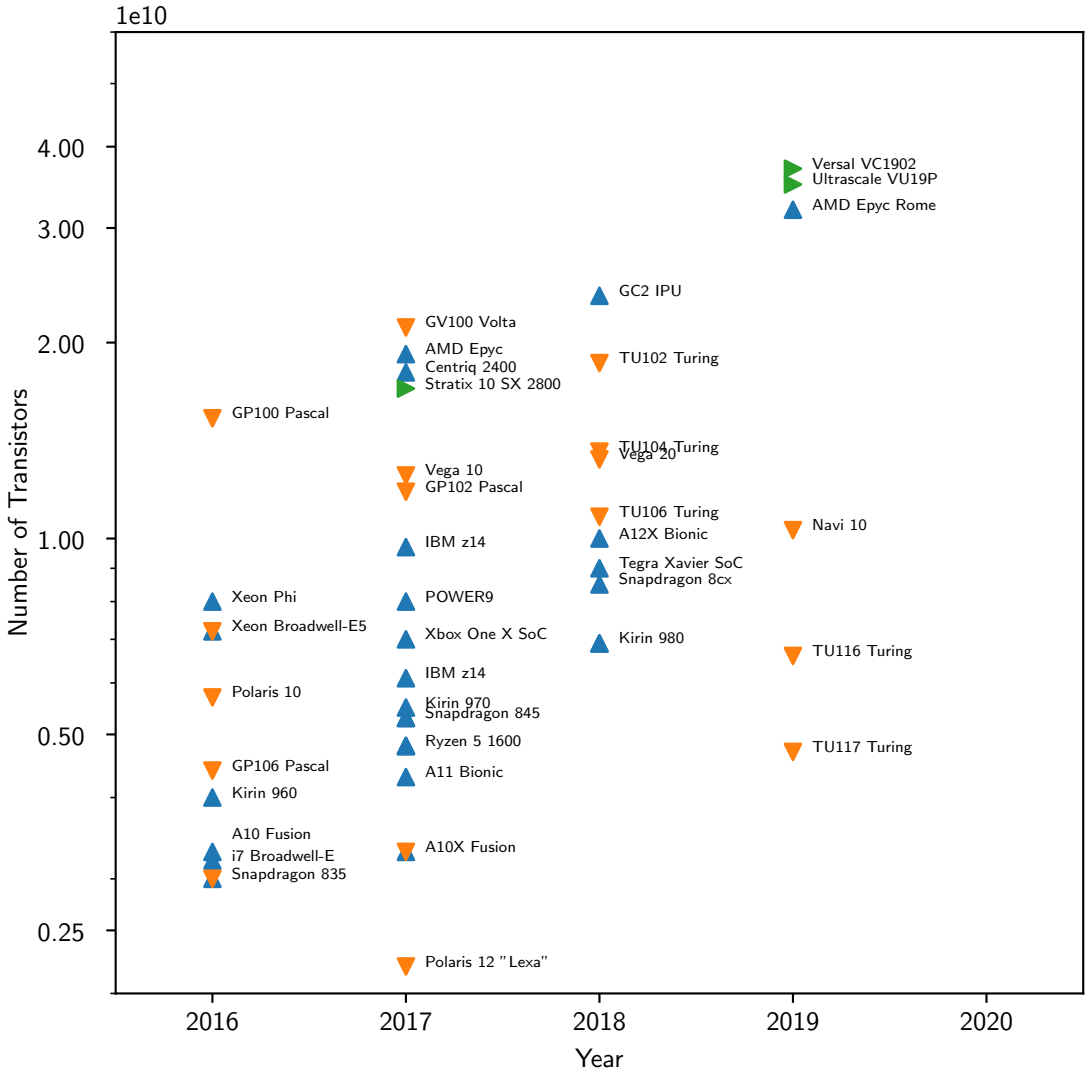


Figure 1.2: Transistor count latest SOCs, data from [1]

resulted in the thrive of more specialized hardware. Traditionally HPC Processors were the leader in demand for high transistor density. The most complex designs today are not only CPUs but also GPUs and FPGAs as seen in figure 1.1 and 1.2.

GPUs are widely anticipated in HPC today and accelerate many applications. In contrast to HPC, AI applications with artificial neural networks do not need floating-point double precision. This resulted in more specialized accelerators like the Google TPU for smaller floating-point formats. Until today it is not clear which architecture is best for inference machines so reconfigurable FPGA-based accelerators became popular.

With the increase of on-chip computing power high bandwidth interfaces in- and outputs (IO) are needed. To achieve the needed IO bandwidth high-speed full custom SerDes Cores are utilized. This results in the complex Mixed-Signal Designs found today. Other needed high-performance full custom blocks are high density and high-speed SRAM based buffers, caches, and other special memory structures.

This increase in complexity in Mixed-Signal Designs have to be handled by the designer. Automated approaches are needed to reduce the design time of these complex designs to finish the design in a sufficient and market-driven time frame. Furthermore, the design and mask cost of these chips have risen meaning the design must be first time right to be economically viable.

1.1 Motivation

From an architectural view, most functional units of SOC designs are digital units. Additionally, there is the effect that digital circuits scale well into smaller nodes, resulting in a larger digital domain in advanced nodes with each node iteration. The digital part is mostly designed in the Semi-Custom flow. One of the key parts of handling quantitative complexity in Semi-Custom Designs is structuring the design and using parameterizable blocks to maximize the re-usability. A high-level hardware description language is used to further accelerate the design process. This part of the Semi-Custom Design is often referred to as the frontend flow.
The backend flow starts with the synthesis step where the high-level design is translated into a netlist and then mapped to standard cells. Furthermore, floorplanning and clocking are set as constraints in the design. With these constraints, the design can be automatically placed and routed.
The Semi-Custom design flow is highly automated which makes it possible to develop digital designs with a low turnaround time between designs and design changes. The resulting circuitry will be a little slower and bigger than a full-custom design but the design time is reduced due to the automation.

Most of today's designs are complex Mixed-Signal designs. Due to the high integration both Semi-Custom and Full-Custom design challenges are highly complex.

In this work, a *Mixed-Signal design* is defined as a mixture of Semi-Custom and Full-Custom blocks. Where simple IO, SRAM, and STD-Cells are seen as the Semi-Custom core elements. This may differ from its traditional definition of a mixture of analog and digital but is a more accurate definition in terms of this work.

Nevertheless today Semi-Custom and Full-Custom designs are still treated differently. Traditionally in Full-Custom designs, each transistor is sized manually and many degrees of freedom exist to achieve the best performing circuit and dense layout. While the Semi-Custom approach is highly automated, the Full-Custom approach lags automation. Unfortunately, the automation does not only lag in circuit design and layout but also in terms of automatically run tests and the verification approaches.

With smaller nodes and larger designs a lot has changed while the Semi-Custom methodology scales, full custom elements are often the limiting factor in terms of time to market. This is further elaborated in the following. While the Full-Custom part architecturally often is a small part of the design it looks different in terms of design time and chip area. This is mainly due to missing automation, reuse, and the suffering from turnaround times and a lot of iterations to achieve the design goal.

In the following two subsections, it will be evaluated how complexity is handled today for large Mixed-Signal designs. For this both Full-Custom and Semi-Custom parts are examined and how complexity changed with smaller nodes in the domain of high-speed design.

1.1.1 Handling Complexity in Modern Mixed-Signal SOCs

As mentioned all of today's largest designs are Mixed-Signal Designs. In this chapter, a modern example of a modern Mixed-Signal SOC is given and discussed. From this SOC the main challenges of today's Mixed-Signal designs will be derived. Furthermore, it can be observed how these challenges are tackled today.

A good example to demonstrate the complexity increase and the handling of it is CPUs. In this case, an AMD Ryzen CPU is used to grasp how complex SOCs are today. Just

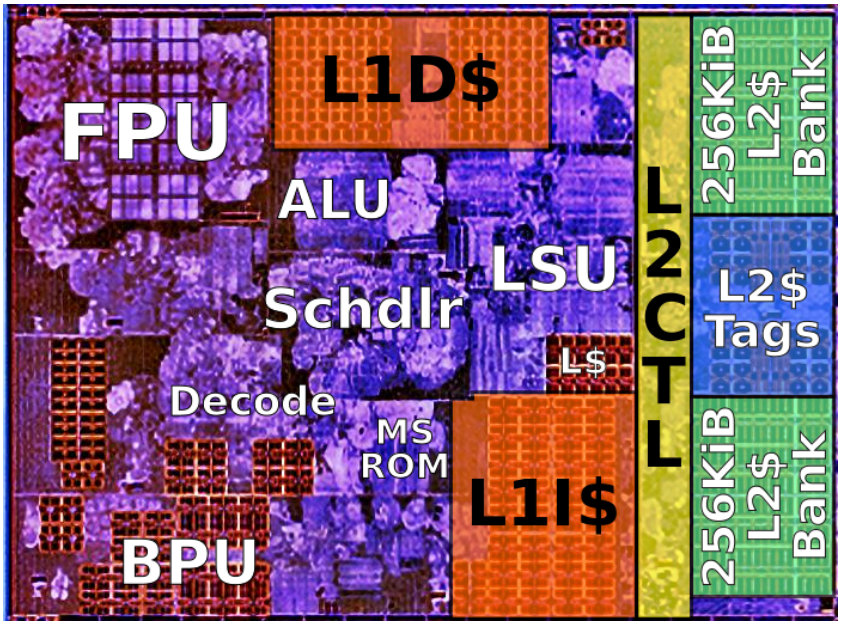


Figure 1.3: AMD Zen Architecture Compute Core from [2]

let us dive into one submodule of the processing die to learn how the SOC is built.

The compute die is the actual processor core which can be seen in figure 1.3. There are some key points which can be observed.
Most of the logic is synthesized with a coarse floorplan, resulting in clouds of standard cells. Working with a coarse floorplan and letting the Semi-Custom flow deal with the detail placements instead of the designer results in reduced design time. With the initial Ryzen CPU, AMD reached a frequency of around 3GHz. While this seems on par with its competitor a more detailed floorplan may result in better performance, the floorplan can be worked on in each processor iteration.
The regular structures are memory or special vector compute units where higher performance or density is needed. Some examples for such units would be cache structures, the Branch Prediction Unit (BPU), or parts of the Floating Point Unit (FPU).

In the AMD Zen architecture, the next bigger structure is the Compute Complex (CCX) which includes four compute cores as shown in figure 1.3 and structures for the L3 Cache.

During the Ryzen development, the technology node changed with each iteration. The core was synthesized for 14 nm GF, 12nm GF and finally 7nm TSMC which is only possi-

ble with low turnaround cycles and technology independent descriptions of the hardware. Initially, the CCX cores were used inside a so-called Zeppelin module. The Zeppelin chip includes 2 CCX cores and High-Speed IO cores to offer the needed bandwidth for the total 8 compute cores. The Zeppelin Core was initially implemented in 14 nm in the next update in the Zen architecture the Zeppelin as a module was ditched. From Zen2 on, two CCX cores and Infinity Fabric links make up a Core Compute Die (CCD). In this architecture iteration, the CCD was implemented with a TSMC 7nm process. In parallel with the Zen2 architecture, an IO die (IOD) exists which incorporates all needed High-Speed IOs. This includes eight Infinity Fabric links, 128 PCIe Gen4 links, and eight DDR4 channels. In the Zen2 architecture, the IOD was still implemented in 14 nm GF.

Up to four Zeppelin Cores are then together on one interposer which completes a Server CPU with 32 Compute Cores. From Zen2 on the Server and Dekstop CPUs are build up from a variable number of CCDs and one IOD. Both interposer configurations with the Zeppelin modules on the left for the Zen architecture and the CCD and IOD configuration since Zen2 on the right can be seen in figure 1.4.

For Zen3 the IOD was ported to $7nm$, while the CCD now only contains one CCX which included double the compute cores. While for the CCD the total number of cores or the cache did not change the eight cores now share the same L3 cache.

One observation is that a lot of Full-Custom Blocks are needed. This is mostly for parts of the design that have to reach a certain density or performance for example all memory structures and the needed High-Speed IO. The High-Speed IO is a complex Full-Custom part of the design. AMD decided to separate the most critical parts of the High-Speed IO on a separate die to separate this problem from the compute cores which consist mostly of Semi-Custom parts. With this separation, AMD can independently scale their critical Semi-Custom circuits on the CCD and their critical IO on the IOD. Furthermore smaller individual die sizes result in higher yields in fabrication. This makes this solution a perfect fit for AMD.

Nevertheless, this approach comes with disadvantages as higher latencies between chiplets. AMD reused the old IO from $14nm$. It can be seen that the scaling of Full-Custom High-Speed IO is one of the main challenges. Other Full-Custom parts as memory structures are normally scaled well into new technologies and are not as critical.

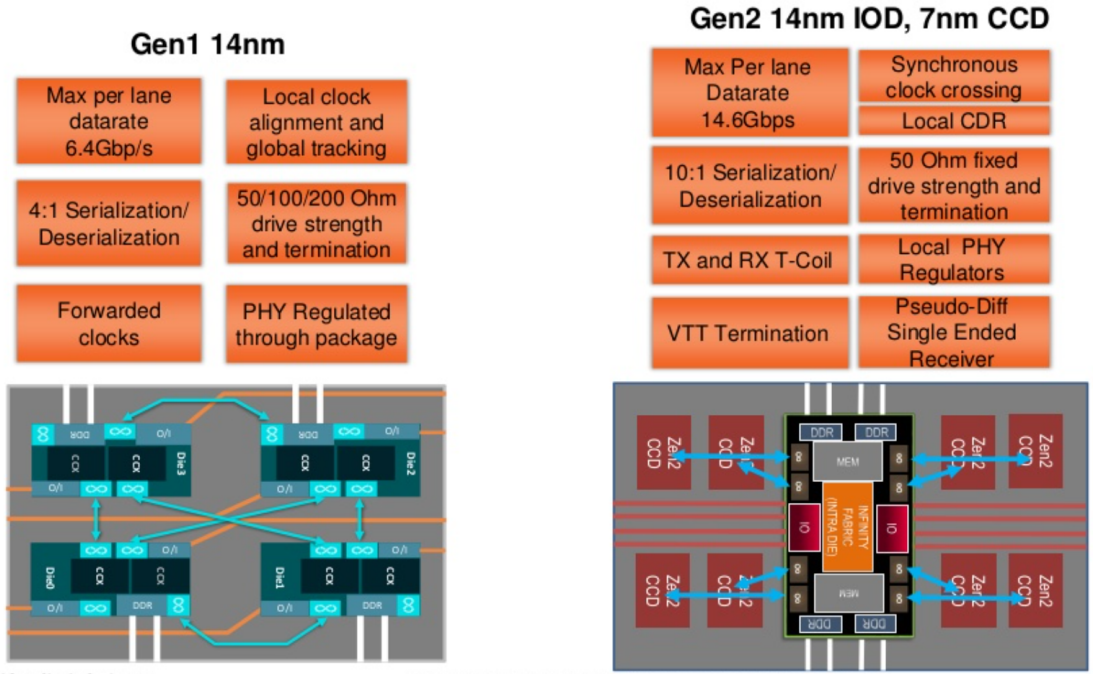


Figure 1.4: IO Changes AMD Zen (left) compared to Zen2(right) [3]

1.1.2 Full-Custom Design in Advanced Nodes

To understand why Full-Custom High-Speed design has such a big impact on design time the main challenges and how it changed from traditional Full-Custom design is discussed in this subsection.
There are many challenges for the full custom domain in advanced nodes. The design process changed from originally being about reaching the best possible figure of merit towards a focus on DFM and manufacturability. Different causes and aspects of these changes will be discussed.

Traditionally the circuit topology was one key factor to decide and various topologies were possible. In advanced nodes, there are many factors especially in the domain of High-Speed circuits which limit the transistor topologies possible. Every additional device added to a high-frequency node adds a parasitic capacitance thus resulting in a slower circuit. Additionally, the available voltage headroom further reduces the choice of specific circuit topology. In today's modern processes the core voltage is sub 1V (i.e. $800mV$ in $22nm$). The voltage ultimately limits the number of transistors that can be stacked efficiently and thus further limits possible circuit topologies.

Furthermore, the low core voltage results in most transistors in the design not being in strong inversion but rather in moderate inversion. The typical square law model does not apply anymore. To handle this modern table based dimensioning techniques like the $\frac{g_m}{i_d}$ method [4] are used.

Since the circuit topology is often quite simple the main focus is on reaching the specified performance metrics for each corner. This becomes increasingly difficult as the variation increases.

One of the reasons for the increased process variation is Pelgroms Law which states that the threshold variation increases with smaller gate area [5]. The derived formula can be seen below where the variation of V_t of a transistor is σ_{VT} and A_{VT} is a constant depending on the process, is increasing with smaller gate area $W * L$.

$$\sigma_{VT}^2 = \frac{A_{VT}^2}{2WL} \tag{1.1}$$

Besides the threshold variation interconnect parasitics have become more prominent. The coupling capacitance of the interconnect and substrate capacitance have become the dominating factor compared to the gate capacitance. This means the layout has a huge effect on the High-Speed circuit behavior and has to be taken into account or mitigated as early as possible. Furthermore also the process variations of these parasitics have increased. Especially in multi-patterned metals and with the needed fill shapes variation and parasitics are additionally increased.

As a result of the increased variation, digital calibration loops are introduced. A calibration loop implemented in a Semi-Custom manner translates the variation to an abstract timing variation for the Static Timing Analysis (STA) and since these calibration loops do not have to run with maximum frequency the timing variation can be simply handled. These loops automatically increase the quantitative complexity since the digital control values and measurements have to be converted in the analog domain and vice versa. For this often ADC, DAC, or TDC structures are needed, which are an architectural challenge and are constructed from unit cells.
A unit cell is a layout of a cell wherewith repetition of this one block the circuitry is build up hierarchically. Typical structures are for example an ADC or DAC, wherefrom a single bit a complete ADC/DAC is build up. These cells are often built for abutment to achieve denser layout results.

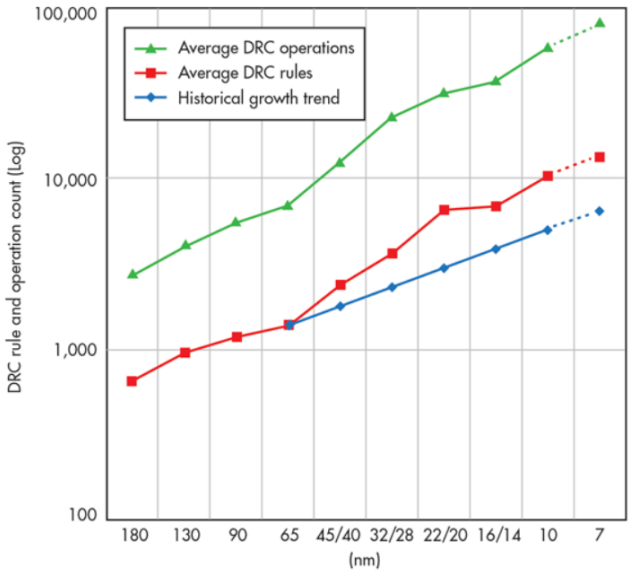


Figure 1.5: Increase of DRC complexity, from [6]

Complexity increases in layout in terms of geometry and DRC too and thus widely differs from traditional Full-Custom design. As for the circuit topology and sizing the degrees of freedom for the layout are also limited. In small nodes, each process step in fabrication becomes more complex resulting in a lot of DRC and DFM (Design for Manufacturing) rules. The design rules for the layout designer to handle have become more complex. To understand the implications some examples are given in the next subsection.

Design for Manufacturing

There are a lot of layout restrictions covered by the DRC rules to reliable produce a modern chip.

The sheer amount of DRC rules significantly increased in the last years. The increase of the DRC rules can be seen in figure 1.5, mind the logarithmic scale. Most simple traditional DRC rules changed into multiple rules with prerequisites and dependencies.

An example is a minimum distance between two shapes of a specific metal layer which originally was a one-dimensional rule. In modern nodes, these rules are multi-dimensional tables depending on the parallel run length of both shapes and the width of the individ-

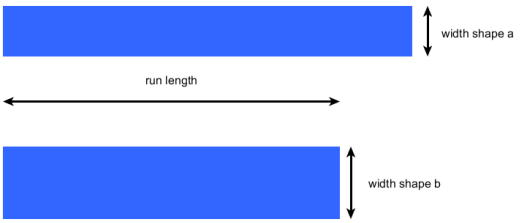


Figure 1.6: DRC Example minimum distance between two shapes

ual shape. This case is visualized in figure 1.6 where two checks need to be done. Shape a with its width and run length and for shape b with its width and the common run length. Depending on this the allowed minimal distance is determined.

Another example shown on the left of figure 1.7 is complex via rules. The minimal poly pitch (distance between two poly shapes) is an important scaling factor. Additionally, it results in a smaller drain-source distance and thus results in very compact via structures since both drain and source have to be fan out to higher metals. In this example, the valid via distance depends on several aspects. In the figure, three configurations can be seen. The left configuration is a DRC error since distance 1 is below the minimal via distance. This seems to be a traditional rule and easy to understand and remember. This becomes more complex in the middle configuration where the via configuration alternates on each stripe. Here the distance is above the mentioned minimum via distance. But there is an additional rule where if more than 2 neighboring vias are below another minimal distance, the via structure is not DRC clean. If only two vias as in the configuration are below this threshold distance 3 and the other neighbors above distance 4, the configuration becomes DRC clean as seen in the third configuration.

These rules make it difficult for the designer to implement the layout with one round trip. It is not possible anymore to keep all these rules in mind while drawing the layout. For this reason, oftentimes only a subset of these rules are used by the layout designer often resulting in larger layouts.
Furthermore, due to this, the number of round trips needed for a good layout increases a lot resulting in longer design times since each round trip in advanced nodes needs a layout change to fix the DRC problem which gets more time-consuming. Additionally, in nodes with many interdependent DRC rules, it is difficult for the designer to fix one DRC issue without introducing new issues. Additionally, this becomes the case when

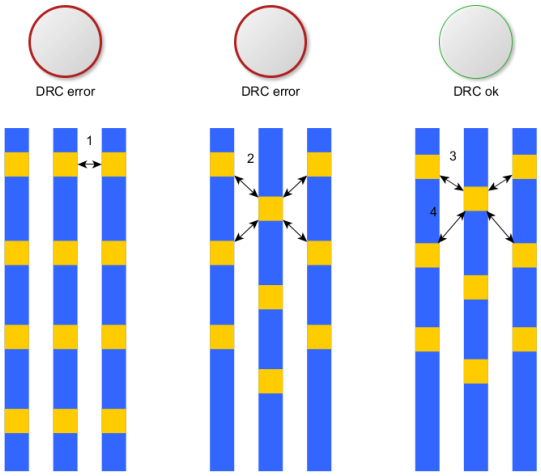


Figure 1.7: DRC Example via distance

several modules will be used in higher layout hierarchy levels and be connected with other modules on the system-level where new DRC issues may arise when best practices were not full-filled. For unit cells, a different problem exists where the individual unit cell will never be DRC compliant but only the whole structure plus special end cells. Which is also challenging. Another problem is the DRC checker runtime which increases with the number of rules and number of individual shapes.

Another aspect of DRC rules is density rules. Uniform density has become an important factor for manufacturing in the chemical mechanical polishing (CMP) step. Reasons for this are that the traditional density rules only state an overall maximum and minimum density requirement which is not sufficient anymore. Accordingly, the rules became more complex. Now there are additional density rules which apply to multiple different sized density windows which are stepped in different sized steps over the layout.
On top of that, there are rules for density gradients between these windows and maximum and minimum accumulated densities over neighboring layers. This further complicates the number of rules and also the number of turn arounds for the layout designer.
Since these rules are based on window sizes they are only taken into account by the DRC when the checked layout part reaches the window size. It is often the case that these density rules are not applied on the module level but afterward when these modules are put together on the system level. This results in a delayed notice of a DRC problem which further increases the turn around time since the later an error is noticed the more difficult it becomes to fix.

To achieve an acceptable yield in advanced nodes additional mandatory rules are defined often called DRC plus. These rules are no hard rules but they rather check the design and give it an overall score. If this score is below a set boundary the manufacturer will not fabricate the chip.

Turn around cycles caused in the late layout step in the full custom design are often the limiting factor in terms of overall design time and time to market.

1.2 Problem Domain

From the above section, two main problem topics can be observed for one design productivity and the High-Speed full-custom domain. The design productivity has to be increased to keep the NRE cost low despite developing complex Mixed-Signal Designs or IP.
Additionally from our example and the observations made in terms of the design productivity gap, it can be shown that a second crucial domain is High-Speed full-custom design which is often one of the most complex parts and the bottleneck in terms of time to market.

In the following, the main challenges should be emphasized. Afterward, the challenges for this work are derived.

1.2.1 Design Productivity Gap

The design productivity gap was first mentioned in an executive summary of the ITRS 1999. Furthermore, in their 2011 summary [7] the design productivity challenge is cited as follows:

> "To avoid exponentially increasing design cost, overall productivity of designed functions on chip must scale $> 2x$ per technology generation."

One of the reasons why this problem arises again and has become a bigger problem now is because it was long mitigated through the popularity and rise of IP Blocks. This resulted in an immense rise of reuse through IP. This can be seen in figure 1.8. It should be noted that the percentage of reuse is based on the silicon area and most macroblocks as SRAMS or IO often are area intensive.

Even with this massive usage of IP, the design productivity gap is still an issue. In a

business report from McKinsey about the semiconductor industry [9] the design productivity gap is described as a destructive cycle together with reduced time to market and product performance. To escape this development trend capabilities must be risen without endlessly increasing design teams. Large teams are costly. Additionally, it is difficult to find additional well-educated ASIC designers on the job market since it is a very specialized field.

In the ITRS 2011 design chapter [7] the overall challenges are described. In general, the complexity increase is separated in two types.

Silicon complexity describes the effect of the technology and interconnects shrink towards Design Technology. The derived main challenges according to ITRS are device parasitics, signal integrity (SI), design for manufacturing (DFM), and decreased reliability. Where decreased reliability includes aging effects for example electromigration. Some forms of silicon complexity were mentioned in the above subsection 1.1.2 especially in terms of DRC.

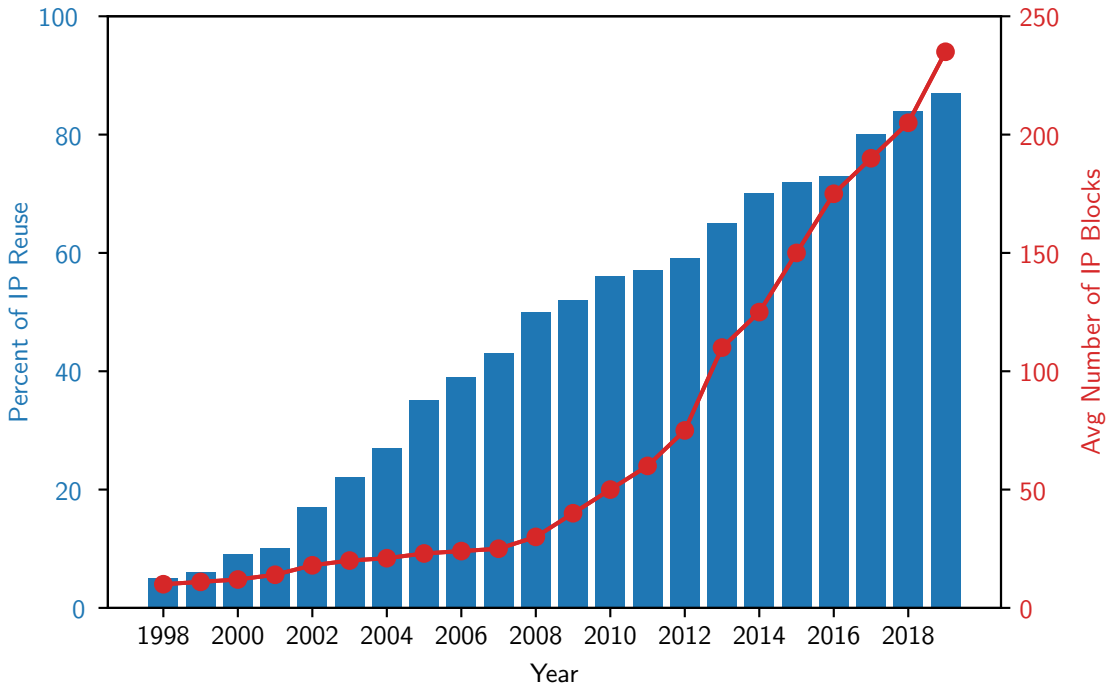


Figure 1.8: Growth in the Number of IP Blocks per Design data from [8]

System complexity refers to Moores Law and the complexity increase through doubled transistor density. This is further amplified by lower cost and shorter time-to-market. Notable challenges for system complexity are reuse, verification and test, reliable implementation platforms, and design process management.

Design Productivity is defined as one of the cross-cutting grand challenges. In the chapter, detailed Design Technology Challenges the rising issues are pinpointed more. In the Design Methodology section, it is noted that especially an automated analog design process is still an open challenge and solutions rarely exist, while in the digital path automation can be seen as the norm.

Further analog, mixed-signal, and RF design was emphasized (chapter *Analog, Mixed-Signal and RF Specific DT Trends and Challenges* in short AMSRF) as becoming a new bottleneck. Analog synthesis methods were predicted by ITRS reports in the past to apply for 25% of all analog content by 2013 and saturate by 2020. In the 2011 report, it is noted that this trend did not occur. There are advances in academia but AMSRF design is still mostly manual. In 2011 the recommendations changed:

> " As a roadmap to analog synthesis has been unsuccessfully placed on the agenda for decades, we should simply let go of the goal of a digital domain-like synthesis process and set more realistic goals."

Reasons for this are the difficult constraints and trade-offs which have to be made in AMSRF design. These tradeoffs and resulting goals are hard to automate as these often rely on experience and knowledge of the component and interaction between all corresponding components.

This does not mean that automation approaches for AMSRF are completely dropped from the focus. The recommendations are shifted towards a more interactive design with semi-automatic approaches.

Furthermore pushing the digital boundary and tuning, calibration through digital logic should be preferred to minimize AMSRF blocks.

1.2.2 High Speed IO

I/O performance has become one of the main bottlenecks of modern computing. This is mainly due to the pin pitch of semiconductors and with it the limited number of pins. To overcome the pin limitation SERDES (Serializer Deserializer) functions were introduced, which have become one of the most important building blocks and core technology for computing. Today High-Speed IOs exist for almost every high bandwidth application.

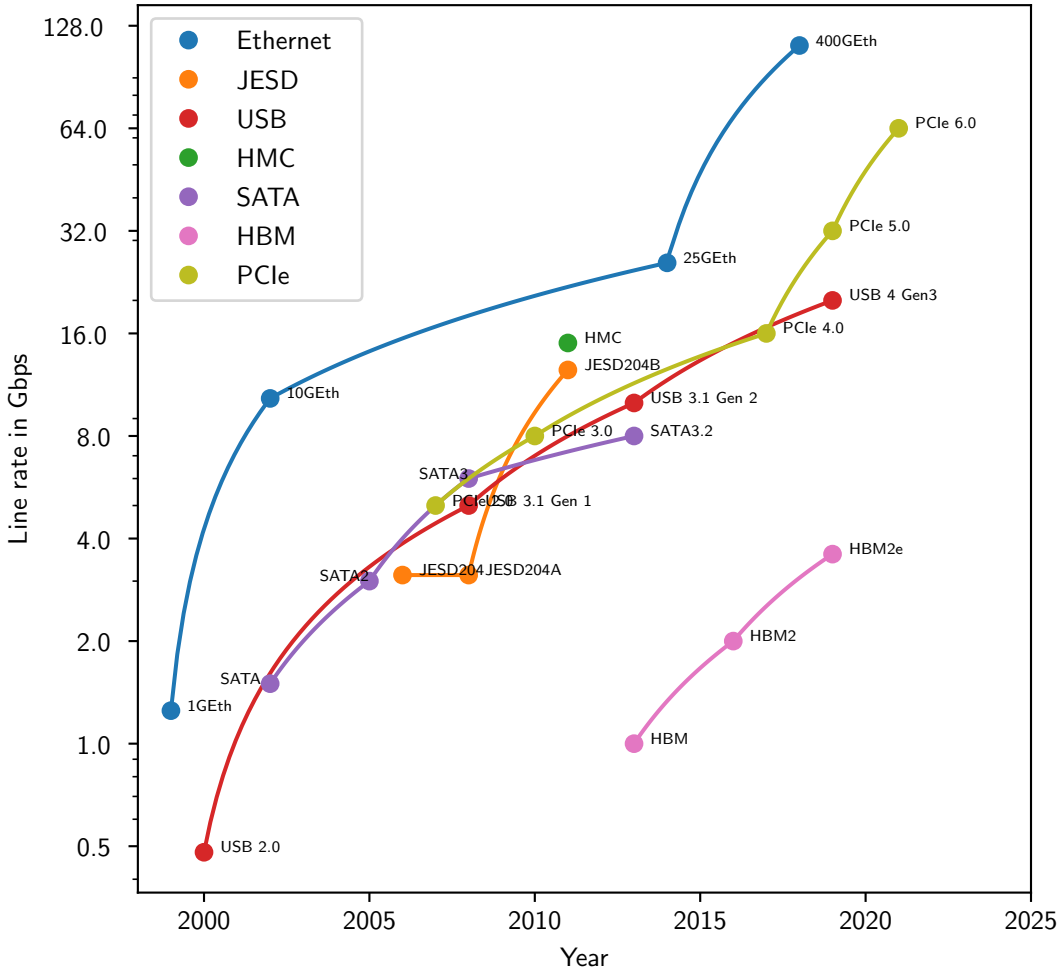


Figure 1.9: Defined line rates of different standards, data from [10], [11], [12], [13], [14], [15], [16], [17], [18]

SERDES data rates have increased in the last years with up to 128 Gbps. Some SERDES line rates for different applications can be seen in figure 1.9. As mentioned SERDES are used in many ASICs today in 1.9 some standards and corresponding line rates are listed. Their applications range from storage (HMC, HBM, SATA), network (Ethernet), data converters(JESD204), internal buses (PCIe) to universal buses (USB).

One aspect to notice is that the data rate of a SERDES is one of the most important and distinguishable specifications. SERDES data rates are mostly driven by standards. It can be observed that over the years and with iterations of these standards the data rate rises.
There is an exemption for this in the plot which is JESD204 and JESD204A where the line rate stayed the same. The reason for this iteration of the standard is that it introduced support for multiple lanes [12].
For Ethernet, it should be noted that not one standard but the highest line rate used today in the standard family was shown. Different Ethernet speeds are implemented in several sub-standards for different applications and different line rates. The given dates are corresponding to the introduction of the Ethernet standard family. This does not mean that at this time the shown line rate was in use. Ethernet is a standard where the SERDES line rates and the corresponding costs are a trade-off between the density of connectors thus resulting in the use of many line rates depending on the application [17].
Most iterations of a standard double the data rate in their next generation. One exempt is the maximum line rate of SATA3.2 which in its maximum mode is called SATAe. SATAe implements a two-lane PCIe3.0 over the SATA connector hence the deviation.
It should also be noted that USB3.0 was renamed to USB3.1 Gen1 and that both are equivalent.

It can be observed that SERDES Cores for different standards have to fulfill vastly different specifications. This is not only the case for data rates as shown in the plot. Depending on the application different channels have to be driven. Their main distinguishable difference is the channel length. This can range from an ultra-short channel as for HBM. Here a storage ic is connected to the ASIC on the same Interposer. In contrast, is Ethernet where depending on the sub-standard the channel becomes long. Channels may be a backplane PCB, a copper cable, or an optical fiber.

This means there is not one SERDES Core to fit them all. Developing one SERDES core for one purpose is often not economically viable since the NRE costs are high and

the potential customers are limited.
One way to solve this problem is SERDES cores which support multiple rates. For standard families, this compatibility to older generations can often be achieved by using frequency dividers. To cover other standards frequency tunable oscillator and circuitry are used. Multi-Gigabit Transceivers are economically great since NRE costs only apply once while multiple standards can be covered. But this comes with area and performance penalties. Since all circuits have to perform with the higher bandwidth and wider tuning mechanisms are needed to cover all rates.

Another point that can be noted is that these SERDES IPs since they have become fundamental building blocks are needed in almost all technology nodes. This also includes the most modern ones. The porting into these new technology nodes is almost always linked with big NRE costs since especially for the full custom parts porting is a cumbersome and mostly manual process.

1.2.3 Derived Challenges

The derived challenge from the design productivity gap is to increase design efficiency. With the domain of High-Speed design in mind, this can be achieved in either increasing reusability or in reducing and eliminating potential error sources early, to reduce cycles in the development.

From the High-Speed IO domain, it can also be seen that reusability is needed but additionally a parametrizable design making it possible to reuse schematic blocks with different specifications. Furthermore, High-Speed IO needs often to be ported into other technology nodes so the approaches should be technology agnostic. Another challenge is that in High-Speed IO design performance in terms of noise, operating frequency, or area is not a possible trade-off.

To reduce error sources each automation step should be in correlation with a continuous top-down flow and consistent. The automation tools should be used in an interaction with the circuit/ layout designer which means the usage should be simple to avoid errors and to reduce the entry barrier to use the tools. Furthermore, layout and schematic automation techniques should incorporate best practices, represent expert knowledge, and thus further eliminating error sources for the interacting designer.

- increase reusability
  - schematic and layout generation
  - no trade off of performance
  - technology agnostic approaches
- reduce and eliminate error sources
  - simple usage
  - correct by construction
  - implement best practises
  - embed in a continuous top-down flow

In the next two chapters first the current State of the Art and how these tools tackle the above-described challenges is examined. Their approaches will be evaluated and their limitations will be shown. Afterward, the new approach of this thesis will be discussed.

2 State of the Art

To keep up with the rising design complexity, the design productivity must rise accordingly. This chapter gives a short introduction to electronic design automation (EDA) tools and their state of the art.

Where in the 70s rubylith operators cut out the chip design by hand (figure 2.1), in the 80s Computer-Aided Design (CAD) was introduced to catch up with Moores Law. These tools developed into the EDA Tools we know today which raised the level of automation further (figure 2.2). EDA vendors release new tools and features each quarter to raise the design productivity but still cannot keep up with the rising complexity.

Design productivity has become a prominent Hot-Topic and the focus of research again. One of the most noticeable research programs is the DARPA Silicon Compiler consisting of two research projects (IDEA and POSH). The DARPA Silicon Compiler belongs to the Electronics Resurgence Initiative (ERI) and is funded with 100 Million US$. The goal of Intelligent Design of Electronic Assets (IDEA) summarized in its kick-off presentation is: [21]:

Figure 2.1: Rubylith operators [19]

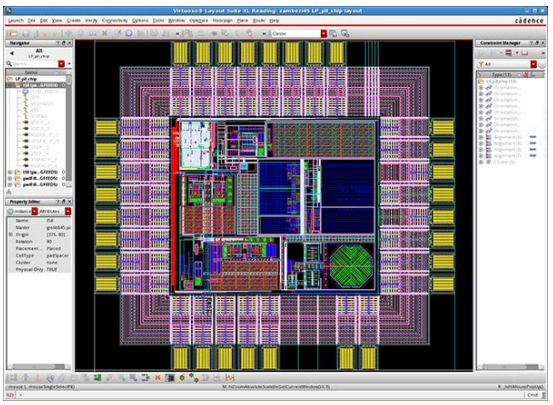


Figure 2.2: Cadence Virtuoso [20]

> "IDEA will create a no-human-in-the-loop hardware compiler for translating source code to layouts of System-On-Chips, System-In-Packages, and Printed Circuit Boards in less than 24 hours."

In this chapter, some State of the Art methodologies and tools are introduced. In the first subsection initially, the general design flow for Mixed-Signal design is described. The focus of this thesis for design flows will be on the full custom part. Some state of the art tools are discussed which are meant to raise the automation in circuit sizing for simulation and verification.
Afterward, there will be a section about different layout generation tools. Three different state-of-the-art tools will be described and an overview will be given. These are the Berkley Analogue Generator, Fraunhofer Intelligent IP, and the PCell Designer from Cadence.

Cadence Virtuoso is introduced and some key concepts of this framework since it is widely used and it is possible to integrate own tooling.

In the last section, a short review and discussion regarding the state of the art tools and research are given. While the ERI projects have in general a similar focus several aspects are left out. Some of these issues will be covered by this thesis, which will be discussed in the last section of this chapter.

2.1 Mixed-Signal Design and Simulation

While tooling is one aspect another aspect is the design flow which defines the general approach in which way the automation tools are used. A tool can support specific design flows or cannot fit. To understand how tools fit into a specific flow some state of the art design flows are introduced.

In Semi-Custom design, a Top-Down flow is used. This means the design starts from its specification, afterwards the system-level model is designed from which the implementation is started.
In Full-Custom design, a Bottom-Up approach was traditionally used. In this approach individual blocks on transistor-level are designed and verified. Afterward, they are connected to a top-level to meet the specification.
These approaches collide in Mixed-Signal design. In a large system, the top-down approach is preferred. From the system level specification, sub-blocks can be derived and

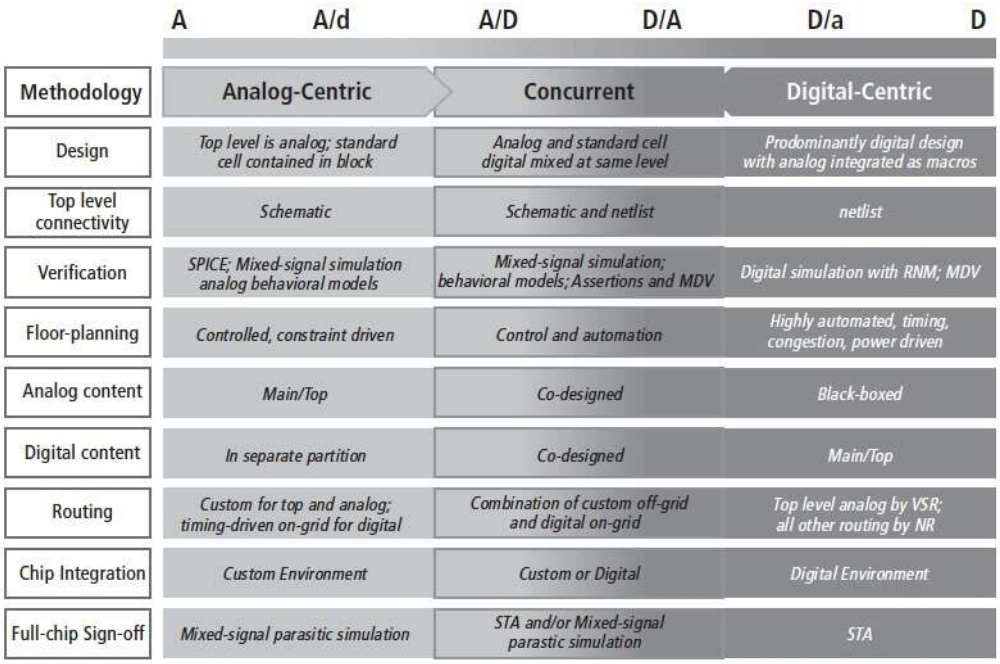


Figure 2.3: Different Design Flows [22]

possible errors can be found early and are not as time-consuming as later in the design process. Designing the Full-Custom part of the chip Bottom-Up in a big Mixed-Signal design is not suitable. Errors on the system level are caught too late and the chance is high that the needed system specifications are not met.

The workflows recommended for Mixed-Signal design by [22] together with Cadence are categorized as Analogue-Centric, Digital-Centric, and Concurrent Mixed-Signal Designs. The different aspects for each approach are shown in figure 2.3. It is stated that the flow should be chosen based on the size of the Full-Custom and Semi-Custom design partitions from big A (big analog part) to big D (big digital part). If the design has no or little semi-custom parts an analog-centric Bottom-Up design flow is recommended. Accordingly, if the digital part dominates the digital-centric Top-Down flow is recommended. For design where the analog and digital part is equally complex a concurrent flow is introduced and recommended in [22].

As mentioned in chapter 1 the Semi-Custom part of complex designs is becoming larger with each iteration. Thus according to [22] a digital-centric flow is most suitable for these designs. An alternative approach to the digital-centric Top-Down flow was implemented

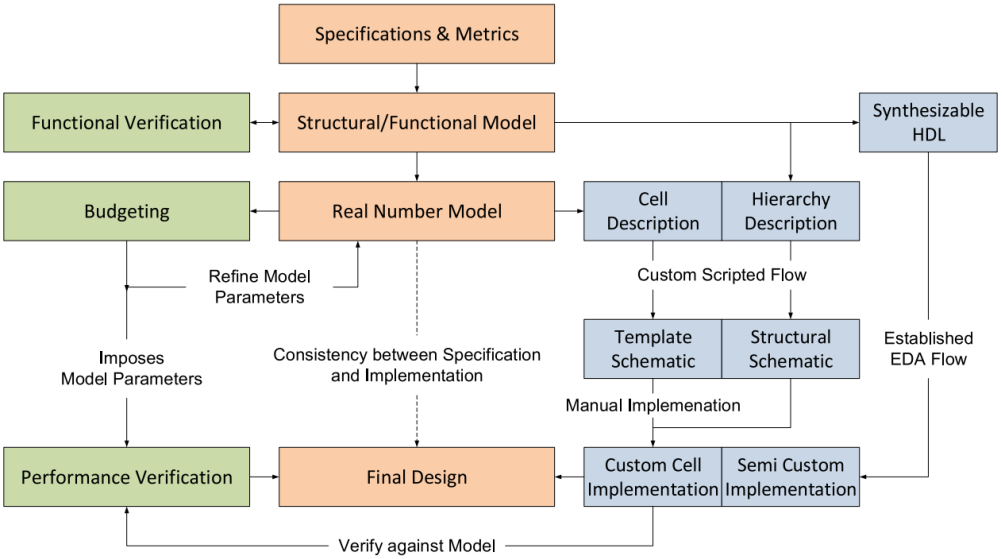


Figure 2.4: Digital-Centric Top-Down Designflow from [23]

with [23] and is since used at the Heidelberg University/ CAG. This flow can be seen in figure 2.4. One of the key differences of this modified flow is that it is a very strict Top-Down Flow.
In this flow the system-level description has to be done first. This system-level module is structured and implemented to a structural/ functional complete model. Each leaf is implemented in a functional Verilog description. Afterward, the analog leaf cells are implemented as RNM models which implements the abstracted behavior of the full custom block. In chapter 6 a reference design is introduced and the module and file structure of such an approach is shown in detail.
This way an executable system-level exists as early as possible. From this structural complete system-level, the digital cells are described with an HDL and developed into synthesizable Verilog which uses the well-known Semi-Custom implementation flow. From the Full-Custom structure and the leaf cells, schematics and schematic templates for the analog implementations are generated. The structural generation of the schematics is done with custom scripting.

A *leaf cell* is a functional block which is the leaf in the tree a structured design spans. In this approach structure and function in design modules are strictly separated.

The *schematic templates* are only a black box where the inputs, outputs, and inout-

put of the analog block are given.

The transistors or passive blocks missing have to be implemented. The leaf cells derived from the system level have to be implemented according to their resulting specification. Since the schematics and schematic templates as black box are generated automatically this flow guarantees a structural consistent implementation and system level. When all full custom blocks are implemented the abstract views, physical description (.lef), and timing information (.lib), are generated for the semi-custom flow to be used as macroblocks. This flow is used in this thesis and especially for full-custom paths some extensions and redesigns were implemented to automate them further. This is described in detail in chapter 6 and 7.

2.1.1 Circuit Optimisers, Expert-Based Sizing

The generation of structural schematics is a good step towards automation but there are still time-consuming tasks. One of them is the modeling of these RNM leaf cells and sizing of the leaf cells. For the implementation of the leaf cells, two tasks have to be done by the designer. Initially, a circuit topology has to be chosen. Afterward, this chosen topology has to be seized accordingly. For circuit sizing tools exists which will be introduced in the next paragraphs and afterward evaluated. To automate circuit sizing two general approaches are often proposed.

Optimizer-based approaches simulate a model i.e. for the highest accuracy Spice or FastSpice of the circuit and try to optimize given parameters to satisfy the given specifications.
In ADE Assembler the Simulation Environment from Cadence optimizers are included. To perform optimization on circuits the parameter space for the optimizer to work on has to be defined. Furthermore, a circuit specification has to be set up to define a goal for the optimizer to achieve.

As an example we take the CML Buffer in figure 2.5. It consists of the differential pair (N0, N1), the passive load (R0, R1), and the current mirror (N2). Additionally, a voltage supply is set up which depending on the corner offers 720mv, 800mV, or 880mV. Inputs for the differential pair are pulse sources with a frequency of 10GHz and an amplitude of 300mV. The voltage offset is $VDD - 300mV$. The differential output is loaded with two small capacitors C0 and C1 with 10 fF.

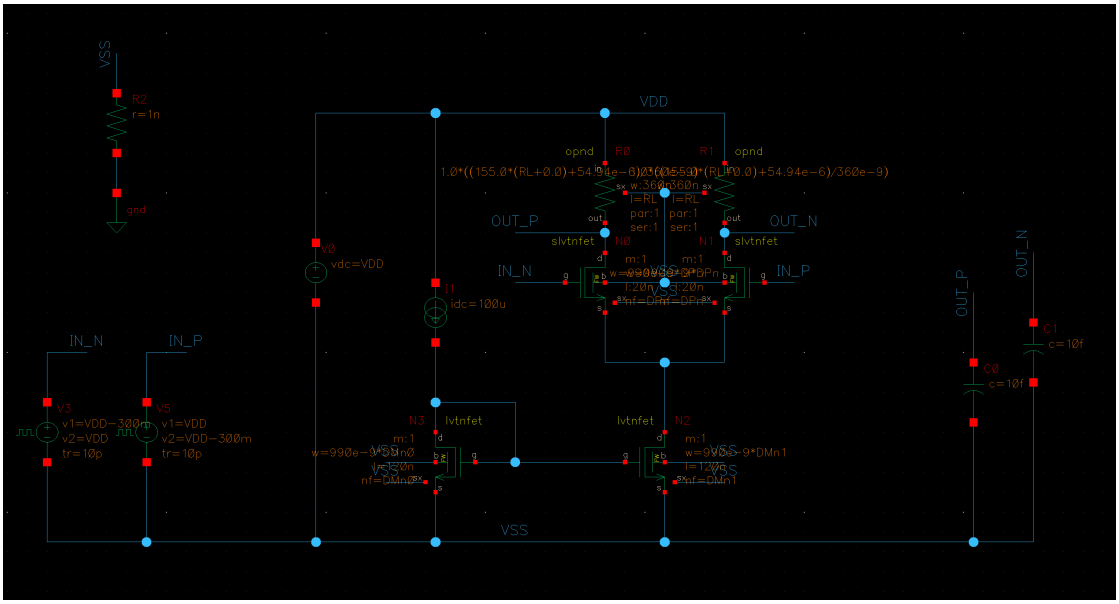


Figure 2.5: CML Testbench as minimal Example

Since the size of the parameter space largely defines the runtime, the goal here is to reduce the parameter space as much as possible in this example. The layout approach at CAG is to work with a handful of standard finger widths for the transistors to achieve a more uniform layout. More about this approach can be found in chapter 6. Furthermore, there are also common transistor lengths for High-Speed diff pairs (minimal length) and standard current mirrors (in this example $120nm$). We use these values to reduce the parameter space for the optimizer. The only variable parameter for the transistors is the number of fingers and furthermore, it is defined that both transistors of the differential pair are matched. As passive load poly resistors are used with a fixed width and a parametrizable length M0, M2 and M3 can vary from 1-25 fingers while M1 can vary from 1-50 fingers the length of the resistor is set between $1\mu M$ to $2\mu M$ with a possible step size of $0.1\mu M$.

With the parameter space defined a testbench has to be set up and specifications have to be defined. The simulation result in ADE Assembler can be manipulated with Calculator/Ocean expressions from the simulation to transform simulated waveforms or values into our desired measurement metric. ADE Assembler also offers an interface to set specifications for the simulation results or Calculator/Ocean expressions. These are then used by the optimizer. For this example depicted in figure 2.5 the output amplitude,

slew rate, and power are used as a specification which then will be used as constraints by the optimizer. The ADE Assembler environment offers to implement different types of specifications namely open-ended, closed-ended, a range specification, and a tolerance specification. Closed-ended specifications are for example lesser or greater than specifications as in this example the amplitude and the slew rate. Open-ended specifications are minimize or maximize. In our example, the power should be minimized. The value is given there is a value that should be reached but the optimizer continually try to improve this even when the value is met.

With this and the corner setup, everything is ready for the optimizer. Each run iteration tries to optimize all defined specifications overall defined corners. For an optimizer run, a stop criteria must be defined this can either be until the specification is met, for a defined number of runs, or a defined runtime. For open-ended specification, the number of runs or defined runtime criteria is more common since it gives the optimizer additional steps to optimize these constraints. A specific number of runs or a specific run time makes sense when either it is unknown if a result can be found or if open-ended criteria exist. In this minimal example, it was configured to stop when the specification was met. This can be seen in figure 2.6. The number left in the table is the run iteration. Iteration 61 found a solution and thus the stop criteria were fulfilled. All other results are the nearest 5 found by the optimizer.

During this minimal example, some observations and notes can be made. Bigger parameter spaces will increase the runtime and thus this becomes one limit for the optimizer. For this reason, the parameter space should be limited as far as possible. To define a reasonable parameter sweep for the individual parameters a good knowledge about the circuit behavior has to exist. This holds also true for the definitions for the constraints since if they are impossible to reach. If the optimizer cannot find a solution depending on the stop criteria this results in either infinitive runs which have to be stopped by the user or maximum runs/ runtime and no solution.

In conclusion, this is for a global optimizer approach often the problem since if the circuit is known well an initial solution can be calculated and the global optimizer is not needed. There exist different ADE optimizers and with an initial solution, a local optimizer or an optimize overall corners can be run alternatively which will additionally reduce the runtime.

Point	Test	Output	Spec	Pre_Typical	Pre_Fast_Lt	Pre_Fast_Ht	Pre_Slow_Lt	Pre_Slow_Ht
Filter	Filter	Filter	Filter	Filter	Filter	Filter	Filter	Filter
Parameters: DPn=20, DMn0=11, DMn1=48, RL=1.8u								
61	PHD_TESTS:OPT_CML_BUF:1	ymax((VT("/OUT_N") - VT("/OUT_P")))	> 250m	362.2m	306.1m	330.1m	388.8m	353.8m
61	PHD_TESTS:OPT_CML_BUF:1	abs(slewRate((VT("/OUT_N") - VT("/OUT_P")) 0 t 2e-10 t 10 90 nil "time"))	> 20G	33.91G	25.41G	30.9G	40.71G	29.81G
61	PHD_TESTS:OPT_CML_BUF:1	average(getData(":pwr" ?result "tran"))	minimize 500u	402.9u	488.9u	488.9u	369.3u	368.2u
Parameters: DPn=25, DMn0=2, DMn1=7, RL=1.4u								
48	PHD_TESTS:OPT_CML_BUF:1	ymax((VT("/OUT_N") - VT("/OUT_P")))	> 250m	273m	219.8m	250.1m	291.9m	303.2m
48	PHD_TESTS:OPT_CML_BUF:1	abs(slewRate((VT("/OUT_N") - VT("/OUT_P")) 0 t 2e-10 t 10 90 nil "time"))	> 20G	26.77G	25.42G	25.02G	32.66G	33.56G
48	PHD_TESTS:OPT_CML_BUF:1	average(getData(":pwr" ?result "tran"))	minimize 500u	349u	410.6u	412.3u	326.1u	326.5u
Parameters: DPn=21, DMn0=13, DMn1=34, RL=2u								
41	PHD_TESTS:OPT_CML_BUF:1	ymax((VT("/OUT_N") - VT("/OUT_P")))	> 250m	259.9m	221.7m	245.4m	267.9m	279.1m
41	PHD_TESTS:OPT_CML_BUF:1	abs(slewRate((VT("/OUT_N") - VT("/OUT_P")) 0 t 2e-10 t 10 90 nil "time"))	> 20G	33.31G	21.82G	28.24G	36.21G	36.31G
41	PHD_TESTS:OPT_CML_BUF:1	average(getData(":pwr" ?result "tran"))	minimize 500u	265.3u	318.2u	316u	246.2u	244.8u
Parameters: DPn=20, DMn0=11, DMn1=48, RL=1.5u								
38	PHD_TESTS:OPT_CML_BUF:1	ymax((VT("/OUT_N") - VT("/OUT_P")))	> 250m	336.5m	271.1m	306.6m	360.9m	361.3m
38	PHD_TESTS:OPT_CML_BUF:1	abs(slewRate((VT("/OUT_N") - VT("/OUT_P")) 0 t 2e-10 t 10 90 nil "time"))	> 20G	29.23G	9.717G	29.59G	33.31G	31.56G
38	PHD_TESTS:OPT_CML_BUF:1	average(getData(":pwr" ?result "tran"))	minimize 500u	402.3u	487.9u	488.4u	368.5u	368.8u
Parameters: DPn=25, DMn0=8, DMn1=36, RL=1u								
50	PHD_TESTS:OPT_CML_BUF:1	ymax((VT("/OUT_N") - VT("/OUT_P")))	> 250m	280.4m	212.3m	248.7m	304.4m	331.8m
50	PHD_TESTS:OPT_CML_BUF:1	abs(slewRate((VT("/OUT_N") - VT("/OUT_P")) 0 t 2e-10 t 10 90 nil "time"))	> 20G	20.76G	18.08G	20.03G	25.96G	32.94G
50	PHD_TESTS:OPT_CML_BUF:1	average(getData(":pwr" ?result "tran"))	minimize 500u	415.2u	501.1u	502.8u	381.2u	382.9u
Parameters: DPn=24, DMn0=23, DMn1=49, RL=1.8u								
47	PHD_TESTS:OPT_CML_BUF:1	ymax((VT("/OUT_N") - VT("/OUT_P")))	> 250m	228m	194.3m	214.7m	236.1m	247.8m
47	PHD_TESTS:OPT_CML_BUF:1	abs(slewRate((VT("/OUT_N") - VT("/OUT_P")) 0 t 2e-10 t 10 90 nil "time"))	> 20G	27.24G	29.85G	29.84G	25.76G	28G
47	PHD_TESTS:OPT_CML_BUF:1	average(getData(":pwr" ?result "tran"))	minimize 500u	225.6u	270.8u	267.1u	210.5u	208.2u

Figure 2.6: Overview of the optimizer run best 5 results

The runtime issue of the optimizer sizing approach can be minimized. The overall runtime can be reduced by reducing the parameter space or losing up the constraints. Another approach is to change the underlying model on which the optimization is run. In this minimal example, a Spice simulation is run within ADE Assembler more abstract simulations can be run i.e. Verilog-A or SystemVerilog. These often do not map to the transistors and thus do not help with sizing but may help to find higher-level specifications.

Another approach is expert-based generators where expert knowledge is decoded and made executable to generate and size circuits. While this approach seems straight forward there seem to be no commercial solutions helping to implement this approach. What is needed for this approach is a sophisticated way to implement the expert knowledge into a parametrizable executable and an interface to the schematic. While commercial tools for this seem to not exist there are several implementations for these generator approaches in academia.

One of them is the Expert Design Plan Language which is developed at the RBZ [24]. In [24] it is discussed that one of the main problems with traditional optimizer approaches besides the runtime is the introduced ambiguity. It is not possible for the designer to transparently understand the reasoning behind the optimizer's found solution. Furthermore as mentioned a detailed knowledge of the system is needed to constraint the optimizer and reduce the parameter space. The argument in [24] is that the designer can use his expert knowledge to describe the circuit behavior in a generic way. For this, the Expert Design Plan was developed which consists of the EDP Language and the EDP

Player. The EDP language is a domain-specific language to implement design strategies and expert knowledge. The task of the EDP Player is to replay the formalized plan with different parameters and from them generate the schematic views.
In [24] some application examples and results with EDP for traditional analog circuits are given.

2.1.2 XModel and Cadence SMG

Another time-consuming aspect is the modeling of the leaf cells especially in the Top-Down flow from [23] modeling the leaf cells. This achieves several important tasks. With these, a first quite accurate executable system exists. From this executable system level, the leaf cells can be updated to meet the system level specifications. When these are met the specifications for the leaf cells become clear. This approach prevents additional turnaround for the implementation since the specification is known and accurate at this point. For RNM or Verilog-AMS modeling some tools exist one of them is Cadence Schematic Model Generator (SMG) another commercial tool is XModel we will discuss both of them and if they will fit into a Top-Down design flow.

Cadence SMG provides a graphical user interface to help build a model from an existing schematic. SMG helps to keep schematic and model interfaces consistent which is achieved by creating in and outputs for the model from the schematic. The model generation is done via the schematic editor where pre-existing model elements can be chosen from a building blocks library. These models are either implemented as Verilog-AMS, wreal, or SystemVerilog. To achieve a high accuracy between these generated models and the schematic implementation simulation results can be mapped via a table model, to the generated RNM/AMS model.
The main issue with this approach is that it is tightly fitted to an analog Bottom-Up approach. The predefined models limit the freedom to custom fit the models for the specific needed behavior. It is important to set a specific abstraction layer to create fast models which accurately model the important needed metrics. Since Virtuoso Advanced M 18.10 (2018) SMG is no longer supported and discontinued.

Another approach is given by the company Scientific Analog [25] they offer several tools to automate and accelerate analog modeling.
Their core technology is XModel which is an extension for SystemVerilog to simulate analog blocks in an event-driven simulator. Scientific Analog claims that their XModel primitives achieve 10x the performance increase compared to standard SystemVerilog

RNM models.
Glister is a tool comparable with Cadence SMG which can generate analog models via a graphical interface. For this similar in SMG predefined building blocks are used. Indifference to SMG these models do not map to Verilog-AMS or a SystemVerilog RNM model but their XModel primitives. An additional benefit brings the tool modelzen which can automatically extract a circuit topology from a schematic netlist that fits a model to it and finally outputs an XModel.
While the approach for XModel is interesting it has to be compared to what the actual speedup of them for abstract RNM models as used in the leaf cells is and if it is possible to use them in the same way. Modelzen is a great approach to create an XModel from a circuit but in a strict Top-Down approach, this circuit does not exist yet. Nevertheless, this could be used to create a more accurate model on which a circuit optimizer might operate to speed up the simulation for these optimizers.

2.2 Layout Generation Tools

The actual layout of an analog design has become a very cumbersome and repetitive task since so many additional DRC and DFM rules have to be taken into account in advanced nodes resulting in more iterations over the layout. Due to this, the analog layout needs increased design time. There are some approaches to automate parts of the analog layout. In this section, a commercial approach is introduced the Cadence PCell Designer, and an academic approach with the Berkley Analog Generator.

2.2.1 Berkley Analog Generator

The Berkley Analog Generator BAG [26] is a framework for analog and Mixed-Signal generators. The aim for BAG is to simplify coding expert knowledge to generate technology-independent, parametrizable schematics and layouts. The BAG framework is implemented in python and can be run with jupyter notebook for exploration-based development.

In figure 2.7 an overview of the general BAG Framework is given. The BAG Generator executes a specific Generator. This specific generator design executes the following in order the schematic creation where the circuit architecture must be chosen and a parametrizable schematic must be implemented. This is based on technology independent primitives provides by the BAG framework. From this, an initial technology-independent testbench is set up and stub circuits are generated. These circuits are

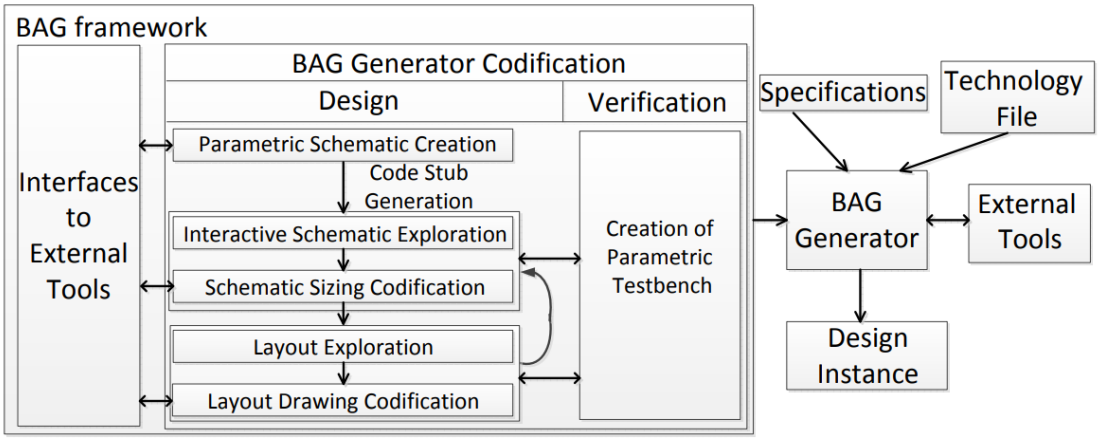


Figure 2.7: BAG Framework Overview [26]

implemented manually by a designer with an initial sizing until testbench criteria are met.
This process can now be replicated by the BAG. The resulting code constructs should also be used in the final generator.
From this, a layout exploration phase follows and the parametrized layout must be implemented. For this similar to the schematic generator first, an initial layout code is implemented. Afterward, with an iterative approach, the code is completed. Furthermore, DRC and LVS can be run automatically. Here the parasitics are available and the schematic can be simulated again with back annotated parasitics to check if the specifications are still met.
For all this, the BAG offers helper classes and abstract methods. Additional BAG provides interfaces to external tooling. As helper classes, a primitive device exists to help with technology-independent characterization. Furthermore to help with common layouts a helper class for variable arrays and trees exists and for a standard cell-based layout style.

A technology-independent generator depends on the developer of the generator while BAG offers an interface for technology-specific metrics. The developer has to use these interfaces to create a technology-agnostic generator.

The BAG was used successfully by Berkley in a DCDC regulator and an LC Oscillator in CMOS 65nm which are given in [26]. Furthermore, perpetrations have been made to easily migrate these designs to CMOS 40nm.

BAG was released under the MIT license at Github but was further developed into BAG2 [27]. The main problem with the original BAG was the circuit verification framework which limited the maintenance and reusability of generators. Furthermore, the original BAG used Synopsis PyCells which are not supported for many modern technologies.

BAG2 adds a universal AMS circuit verification framework and implements two new layout engines which allow technology independence in a more sophisticated way. BAG2 was used to implement more complex generator examples as a SAR ADC and a SerDes transceiver frontend. BAG2 has successfully been used in several technology nodes as TSMC 28nm, 16nm and GF 45nm RF-SOI, 22 nm FDX, and ST 28nm FD-SOI.
The design flow of BAG2 is similar to BAG but provides additional API features for the schematic generation i.e. manipulating pins, instances, and wires. In parallel, a testbench can be generated with the universal verification framework which decouples the testbench description from the specific circuit to be able to reuse and maintain more general testbenches. For this clear interfaces are defined over which the testbench can be described in a tool-independent manner.

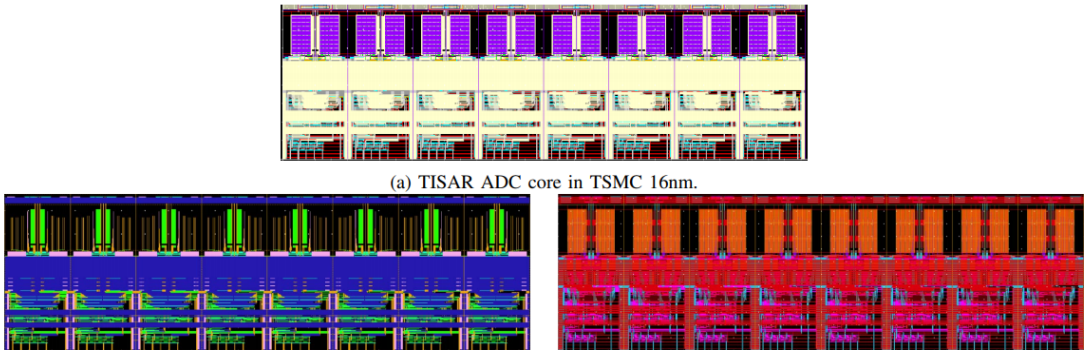
For the layout generator, one of the new layout engines can be chosen. The XBase layout engine provides abstract python classes which encapsulate common layout methods and specific primitives to separate them from the technology-independent generator. Furthermore, XBase supports the layout styles from the original BAG.

The other new layout engine is laygo (LAYout with Gridded Objects) which is meant to place layout elements on grids and uses relative information instead of absolute coordinates. Different from XBase laygo does not use programmed primitives but rather manual layouted ones. Process independence is given by assuming that the manual base layouts are created with the specific grid in mind.

In figure 2.8 the result of using both laygo and XBase for time-interleaved SAR ADC core is shown it was generated for TSMC, ST, and GF.

2.2.2 Fraunhofer Intelligent IP Framework

The Intelligent IP Framework [29] is developed at Fraunhofer Institute for Integrated Circuits. It is another academic procedural generator approach for analog circuit design.



(a) TISAR ADC core in TSMC 16nm.

(b) TISAR ADC core in ST 28nm FD-SOI.

(c) TISAR ADC core in GLOBALFOUNDRIES 22nm FDX.

Figure 2.8: Different generated TISAR ADC layouts [28]

The framework is based on python and offers an object-oriented API for the user to write generators. IIP generates all views needed in a design flow. IIP is a proprietary tool and thus first-hand experience could not be made. The analysis in this subsection and the review is based on publications made for IIP.

The IIP implements a Technology Abstraction Layer (TAL) to enable highly generic generators. These generic Generators should be usable in other design projects and also other technologies.

Another important part of IIP is the abstraction of design environments allowing the generator to interface with different tools.

Generators are implemented as a class that inherits some methods and properties. From inheritance, some basic methods exist which have to be implemented. With param_spec() parameters are defined. Next param_check() allows considering interdependencies between parameters. Several more methods exist to prepare the values for the generator and to draw the schematic, layout, and symbols.

In [29] the generator was successfully used to implement different current steering digital to analog converter in several technologies i.e. 180nm bulk and 28 nm FD-SOI. The design was taped out in 28 nm.

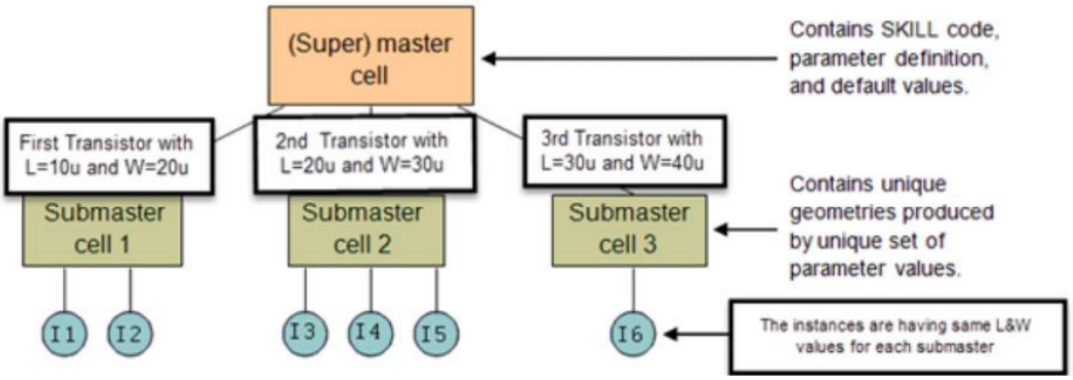


Figure 2.9: Pcell internal structure master, submaster from [30]

2.2.3 PCell Designer

A different visual programming based generator approach is the PCell Designer from Cadence.

PCells are the parametric cells inside Cadence Virtuoso which are based on the SKILL programming language. Both PCells and SKILL are proprietary. Still, PCells have become a de facto standard and are included in all PDKs.
Alternatives are offered by Synopsis with the PyCell environment for which PDK views from some big foundries exist.

Traditionally to create a PCell SKILL can be used. A detailed introduction for this can be found in [30]. The special SKILL method pcDefinePCell creates the PCell. pcDefinePCell creates the so-called superMaster cell which is an instance with the default parameters of the PCell as seen in figure 2.9. The compiled SKILL code is then attached to the superMaster cell. If an instance is created from the PCell, the environment first checks if the configuration exists if it does not exist a submaster is generated. Instances with the same configuration will share the same submaster which is a derived copy from the superMaster.

```
1 pcDefinePCell(
2 list( ddGetObj("myLib") "myCell1" "layout")
3 list((w 0.2) (l 0.1)) ;; Formal parameters of PCell
4 let( (cv)
5  cv=pcCellView
6  dbCreateRect(cv list("Metal1" "drawing") list(0:0 w:l))
7 ) ;let
8 ) ;pcDefinePCell
```

In the above listing, a minimal PCell example implemented in SKILL can be seen. As the entry in line 1, the pcDefinePCell method is used. The first parameter of the method in line 2 is the library cell name and cell view where the superMaster will be generated. The second parameter in line 3 is a list of keyword value pairs which will become the parameters and default values of the PCell. The last parameter beginning from line 4 to 7 is the generator code that will be executed.
pcCellView in line 5 receives the cellview on which the generator will work. In line 6 a rectangle is generated on layer Metal 1 starting at origin (0,0) with the parametrized width and length.
The PCell can now be compiled and used.

There are several limitations for PCell code there are solely methods allowed from the db, dd, cdf, rod, and tech subsystems and the pc Skill methods since the translators who convert the code to a layout can only operate on these functions. These translators are called for example during netlisting of a schematic including a PCell or for streamout during a DRC or LVS run.

The allowed commands are primitive layout/parameter/common data format interfaces and with this writing complex PCells in SKILL needs a lot of code to achieve a high-level description for generators.
A high-level description for PCells is offered by the PCell designer. Here additional more abstract methods to query and to create shapes are added. Furthermore, the PCell Designer offers an object-oriented approach. In the PCell Designer the PCells are programmed graphically.

In figure 2.10 the PCell Designer Environment is shown. The PCell Designer can only be run within Virtuoso and presents itself with three main windows. The command

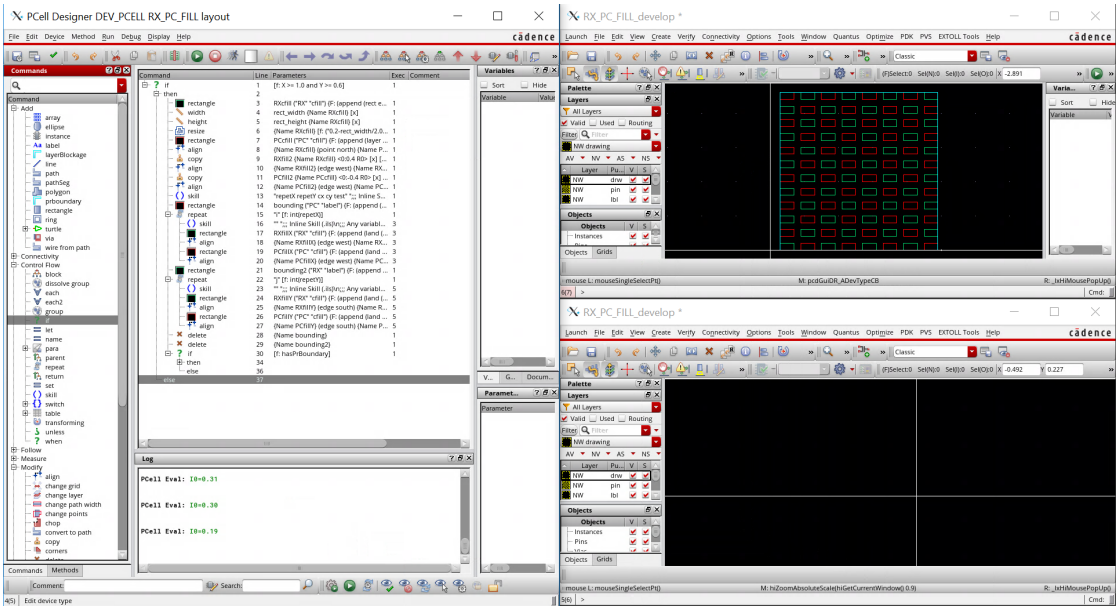


Figure 2.10: PCell Designer environment overview

window on the left where the graphical programming takes place. A layout preview window where this code can be live visually debugged. And another layout window here on the bottom right from which manual shapes or instances can be imported into the command window.

A typical workflow to create a PCell with the PCell Designer can begin with some initial manual drawn layout shapes or instances. These can be imported into the PCell Designer command window and later be manipulated. The Parameters for the PCell are given over an extra menu entry where different data types, alternative descriptions, and a range for a value or a set can be given. In the main window, the generator script is build up for this different graphical representation of commands, loops, and conditions can be set up.

One concept which helps with efficient querying of shapes and preparing new ones is so-called geo expressions. The idea here is to use geometric expressions which can be nested to create a selection or shape from which can be drawn or which can be used as a reference. An example of such a geo expression which gets the source shapes of a PCell transistor instance over a vertical metal shape on metal 3 is shown in figure 2.11.

The geo expression first selects the needed instance I0 seen at a) afterward with the

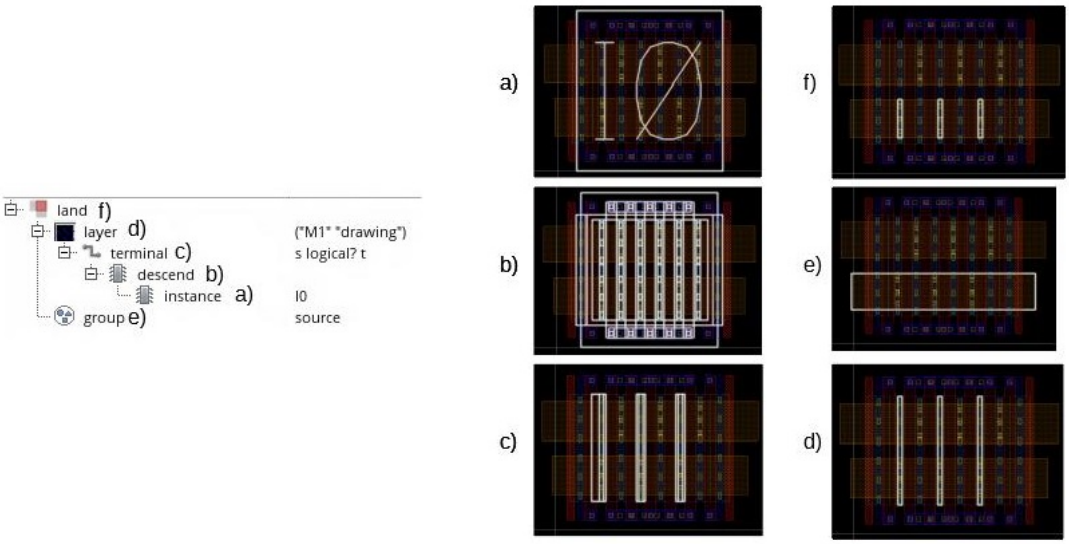


Figure 2.11: PCell Designer example geo expression

descent command at b) all shapes inside the instance are queried. These shapes are filtered by command c) a specific terminal in this case s for shape. All source terminal shapes are now selected this does not only include the metal terminals but also some active shapes. To further filter these with d) the wanted layer metal 1 is selected. In parallel we select with e) our metal 3 stripe we want to connect to. In f) we combine both selections with an intersecting set.

In the graphical interface methods of a PCell can be defined and it is possible to let PCells inherit from each other. On top of the standard skill interface additional more powerful methods are given to the user to efficiently create vias, manipulate PCell instances and their parameters and draw more complex shapes.

A more elaborate example of how PCells are designed with the PCell Designer is given in chapter 8 where one approach for layout generation is discussed in which the PCell Designer was used. The above-shown geo expression is from one of these PCells and it is further used to create vias over the resulting list of geo shapes.

An additional feature of the PCell designer is that it is also possible to create so-called AppCells, which are not compiled as PCells but rather a way to graphically write SKILL

programs working on layout, schematic, or symbol views without creating PCells.

2.2.4 Review Layout Generator

Each layout generator discussed here has some advantages and disadvantages. While the BAG is an open-source project all other examples are proprietary. Out of the proprietary generators the Cadence PCell generator could be tested since the University of Heidelberg is a member of the Cadence Academic Network Program. Fraunhofer IIP was evaluated based on their publications.

While the BAG2 generator framework yields good results as shown in [28]. The main problem of this approach is on the one hand a very complex setup where many technology descriptions have to be created. The BAG and its layout generators and tutorials are distributed over several GitHub projects, where several are legacy and some not supported anymore. The general overview of the project is difficult to grasp since the online documentation is also not up to date.

On the other hand, the primitive cells needed for the XPath layout engine are technology-dependent and have to be set up for each technology. These primitives for example transistor primitives can be very complex. Especially in advanced nodes, shorter transistors have a lot of additional rules on how they can be drawn. This includes additional dummy structures, different drain-source distances for several lengths, and more. In [31] BAG was used to automate some layout aspects of a DDR PHY Design. Here only transistor sizes above 70 nm were implemented in a 22nm technology as primitive to keep the complexity in check since at this size no additional rules are applied anymore.

Another aspect is the integration into a Top-Down Mixed Signal Flow where the analog implementation is derived from the system-level model. While it is mentioned that BAG will integrate well in Mixed-Signal design the approach for this is a very separated approach. At Berkley Chisel is used as a hardware description language for digital logic and python with the BAG for the full custom approach. Both are treated separately and do not provide a consistent flow.

The PCell designer is a generator approach that seems to eliminate the troubles of writing a PCell purely from primitive Skill functions and embeds well into the layout step since their PCells can be integrated easily. A more detailed analysis of the PCell designer and its disadvantages can be found in chapter 6 where one of the approaches

of this work is based on PCell generated with the PCell Designer.

2.3 Cadence Virtuoso

Cadence Virtuoso is one of the major tool environments for Full-Custom design. The Virtuoso design environment incorporates a collection of tools needed in the full custom domain. This includes schematic entry, layout editor, simulator, and more. In this section, the basics of this environment are introduced. Later approaches of this thesis will interface with Virtuoso so the most important concepts of this platform and some definitions will be done here.

The Virtuoso main window which appears on start is the so-called command interpreter window CIW. From this window, sub-tools can be started for example the already mentioned PCell Designer or Cadence SMG. Furthermore, the Library Manager can be shown from which the full custom design can be managed. The CIW as the name suggests can be used as SKILL command interpreter to run SKILL commands or load SKILL scripts into the active environment. Nearly every subsystem offers a SKILL API and can be scripted.

The design is managed by the Library Manager. Here different libraries are shown. One of them is the vendor PDK library. In Virtuoso a *library* is a set of cells. While cells can have several cellviews. A *cell* represent a design element. The cell can include different *cellviews* of the same design element in the design hierarchy. Cellviews are the different representations of a design element. This can be the layout, schematic but also more abstract views (SystemVerilog), simulation views (maestro), and or config views (config).
Cellviews are defined with a cell name and a cell type. The cell name can be chosen freely, the cell type is used as information for the Virtuoso environment. Depending on the cell type different applications are chosen for the cellview object.
CDFs are parameter templates that are shared between different cellviews. Each cellview individually copies this template initially and then changes the default parameters to unique parameters to the cellview. Besides the cell parameters, the CDFs also include modeling information (terminal order, model, name) for different views and callbacks function calls.
While each cellviews share the CDF view these cellviews are still completely independent. There is no common data structure enforced the only thing they share are these

CDF templates. But since both views are completely separated this does not mean that both have to have the same instances in and outputs or parameters. This may result in potential inconsistencies between cellviews which may be discovered late in the design. This is a potential error source that should be considered.

The most important sub-system in virtuoso is schematic entry to create schematic cellview of a cell, the layout entry to create the layout cellviews, and ADE Assembler/- Explorer. Assembler/ Explorer are simulation environments in which simulation setups can be done, a simulator can be started and analysis can be done.

Since 2018 Virtuoso exists in two different versions IC and ICADVM where ICADVM is the version used to work with advanced nodes. ICADVM offers features needed in these nodes in example coloring. ICADVM and IC are otherwise compatible but differ in licensing and additional features for advanced nodes.

2.4 Overall Conclusion

The introduced tools offer or promise further automation. The main problem is mostly that they are often not suitable for bigger Mixed-Signal designs since they won't fit in a strict Top-Down flow. Either because views needed do not exist yet in the flow as for Cadence SMG or glister where a model can be generated from the schematic but in a Top-Down flow this model is needed first and the schematic should be derived from it. The case is that they offer a completely separated approach from the system level, which is the case for BAG. In these approaches system level and implementation are developed separately and introduce an error source for potential inconsistencies.

Virtuoso is a framework for Full-Custom design that is highly modifiable and extensible with SKILL. Every subsystem in the virtuoso environment can be automated. The concept with different independent cellviews is versatile but introduces potential consistency issues. The consistency between different cellviews is checked at some points for some aspects for example inputs and outputs of symbol and schematic which is checked at save or the connectivity engine within some versions of the layout entry where instances and connections are checked live. These consistency issues have to be taken into account when working or extending the Virtuoso framework since introduced errors may be found late with the LVS step and result in time-consuming fixes.

The above-mentioned DARPA research projects seem promising and already yielded some results. One part of this is the OpenROAD project which can be found at [32]. It is an open-source project targeting the goals of the DARPA IDEA program (24 hours "no human in the loop" design). Its goal is to implement an academic reference flow for the goals set in IDEA.

In [33] the state of the project is described after six months of runtime. During these six months, several already existing open-source tools have been developed further and new projects were started to achieve a first digital RTL to GDS flow. These projects include static timing analysis, logic synthesis, floorplan, and power distribution network, clock tree synthesis, placement, and routing. Furthermore, research was done in terms of using machine learning to self-drive the OpenROAD flow.

In OpenRoad the main focus lies on a digital RTL to GDS flow and running it with the help of machine learning. Full-Custom design seems only to be viewed as abstracted macros thus leaving this problem field untouched. The only open-source tool in OpenRoad for Full Custom Design is Magic which is a VLSI layout tool written in the 1980's [34]. It seems mostly to be used to visualize steps in the digital flow.

3 Improving Design Efficiency in Full Custom Design

In this chapter, the extend and goals of this work will be defined. For this first, the problem domain of High-Speed advanced nodes design and the existing design productivity gap will be analyzed. Afterward, some general design approaches are derived from this analysis. In the last section, the resulting goals for this work and the derived methodology are introduced.

3.1 General Approaches

In this section, some general approaches which should be taken into considerations and do not need special or new software implementations are noted. These points should always be considered when designing full custom blocks or adding custom tools for design automation. Two main topics will be discussed which are, on the one hand, semi-custom and full-custom border and on the other hand consistency between different design views.

3.1.1 Semi-Custom and Full-Custom blocks and border

As above noted AMSRF circuitry should be minimized as much as possible. The amount of effort of a full custom implementation is linear to its unique circuits [35]. As already mentioned one way to reduce the unique circuits is moving as many elements as possible in the Semi-Custom Domain.

With each new node, the digital boundary can be pushed further since the maximum operating frequency for the STD-cells will increase as the nodes become smaller. Additionally, the Semi-Custom flow is mostly technology independent which means these circuits can be ported with a low effort into a new technology node.
This is especially the case for High Speed or RF circuitry where the decisions for this boundary are often defined by the maximum operating frequency of the Semi-Custom part. While shifting the digital boundary some side effects have to be taken into account.

Semi-Custom Circuitry is noisy and has to have its digital supply. This means signals have to cross power domains at defined interfaces. Additionally deciding to implement a sub-block in the Semi-Custom domain instead of Full-Custom can be a design trade-off between time-to-market and performance. In the Semi-Custom domain, not all performance parameters can be controlled as well as in Full-Custom for example phase noise or corresponding slew rates.

On the other hand, there are a lot of cases where a lot of functionality can be implemented as Semi-Custom solutions for example calibration loops and general control logic. These approaches introduce new analog blocks needed in the design to convert digital signals into the analog domain or analog in the digital. These can be or include ADC, DACs, and TDCs. While these analog circuits are added they have the advantage that they are often build up from one unit cell and highly structured. An additional advantage is that the integration of digital calibration and control logic enables programmability of these since configurations and measurements can be included in the chips register file and made accessible over the software interface of the chip.

3.1.2 Consistency of Views

Design inconsistencies are one of the main issues which can introduce subtle erroneous differences between design views that may not be caught. There are some best practices with which these errors can be eliminated or at least made very unlikely.

In our design flow, the schematic hierarchy for Virtuoso is generated from the executable system-level description. With this, the generated implementation hierarchy and schematic template interfaces are consistent with the system level. This generation is only possible in one direction from system-level to implementation. Thus the generated structural schematic and interfaces should not be modified as with these modifications inconsistencies are introduced. Rather the system level should be modified and the implementation regenerated. In a one-way generation approach, the generated result should always be treated as read-only.

With this approach, most inconsistencies can be avoided. Possible error sources left are the interfaces. While the interfaces of schematic templates are generated the schematic content is created by the designer. This can lead to misinterpretations and little errors in example for signals where different phases are needed. To reduce interface misinterpretations or errors complete signal paths should be simulated in a Mixed-Signal Spice/Verilog

simulation as often as possible. In these simulations, not the complete behavior has to be simulated its main purpose should be to check the interfaces and signal forms.

3.2 Derived Methodology

From the above-stated problem domain and the general approach, the methodology presented in this work and the goals for the automation tools are defined. The overall goal given from the design productivity gap is to rise design efficiency.

A common problem in creating silicon IP blocks is to offer customer-specific solutions. Such specific solutions often require modifications to the full-custom part of the design. These changes can include reaching different performance metrics or line rates but also could mean implementing the core for different stack-ups, different technology nodes, or with different lifetime and or resilience constraints. These changes often trigger a chain of modifications on the layout and schematic end.

Especially for SERDES IP with all the different specifications modifying an already existent IP is often needed.
A goal of this work is to accelerate the design of full custom blocks and also reduce the work which needs to be done if either a new specification must be met, the technology node or stack-up changes.

To automate full-custom design decisions, schematics and layouts an expert-based approach is chosen where an executable expert description of the schematic and layout or parts of the layout exists.

From the lessons learned from the ITRS report, these solutions will be used early in the design and layout generation of real-world designs. For this, especially for the layout generation some of the toolings were applied to industrial designs in industrial cooperation to prove practicability.

The main problem in analog design which has to be tackled is the linear increase of effort with each unique circuit. In this work, the number of unique circuits will be reduced by adding parametrized circuit generation for schematic and layout.

The second approach to improve design efficiency through minimizing turnaround cycles

for the designers. This will be reached if these generators can be seen as correct by construction. Thus the usage of the generators will reduce and eliminate possible error sources. To achieve this testing methods have to be applied.

There are also as many technology-agnostic solutions needed as possible to enable porting into new technology nodes or other stack-ups as effortlessly as possible.

The implemented solutions in this work are modular and thus can be separated into the following categories:

3.2.1 Schematic Generation

The *Schematic Genration* focuses on the parametrized generation of leaf cells and should complement and extend the strict Top-Down approach from [23]. For this, the schematic leaf cell generation must be attached to the structural generation from the system level. This ensures that all parameters are propagated and consistency is kept.

Another important aspect is the expert description itself which has to be simple and allow an easy-to-use interface to generate schematics and size them. For sizing table-based approaches as the $\frac{g_m}{i_d}$ method should be supported. The global propagated and local parameters in the system-level description have to be used as high-level specifications from which the sized circuits can be derived. The solutions implemented for this are detailed in chapter 5.

3.2.2 Layout Generation

The goal for the *Layout Generation* is similar, for one reducing the number of unique layouts which have to be implemented and on the other hand reducing potential error sources to minimize turn arounds.

Creating an efficient executable parametrized layout generator is challenging especially with reuse, reliability, and maintainability in mind. It was quickly observed that the leaf cell layouts are build from a lot of repeatable layout structures. Some of these layout structures are identified in this work and implemented as so-called *elementary cells* This achieves two advantages first the problem is divided into smaller chunks which relaxes the resulting above mentioned problem. Another beneficial aspect is that it results in an automatic rise in code reuse and a more uniform layout. An initial implementation

of the elementary cells is given in chapter 6. This concept was initially implemented with Cadence PCells. These had some major disadvantages and a technology-agnostic approach was implemented with XCell.

The XCell approach is very interactive allowing the designer to constraint parts of the circuit for partial layout generation. When the layout designer generates a partial layout, this layout must be a good starting point. The layout generators should implement best practices in terms of integration and DFM. For DFM these layouts should incorporate the generation of dummy structures and fill structures.

In modern technology nodes especially with multi-patterning, the lower metal layers and the doping layers often have a lot more DRC rules than the normal higher metal layers. This makes these layers hard to use since many rules have to be taken into account. The goal for the partial layout generation should be to focus on these layers to reduce error sources for the designer and thus decreasing iterations over the layout.

4 Advanced Node Design

In this chapter, an overview of the technology nodes used in this work is given. These include a 28nm bulk node and a 22nm FDSOI node. First of all, an overview and comparison of bulk and FDSOI are given. Afterward, an overview of electromigration will be given which has become an increasing issue. Both topics will be introduced and discussed here to create a common ground for topics in chapter 6.

4.1 Bulk and FDSOI

The methods introduced in this work will have to handle different technologies. To understand the differences which have to be handled two modern commonly used technologies are here introduced. In this section, we will compare a 28nm bulk node and a 22nm FDSOI node. Especially FDSOI has some beneficial properties which can be used and have to be understood.

In figure 4.1 both bulk and FDSOI CMOS cross-sections can be seen and compared. On the left side, the bulk CMOS cross-section can be seen. The substrate is weakly p doped and connected via a heavily doped P contact crating the bulk of the transistor. Two

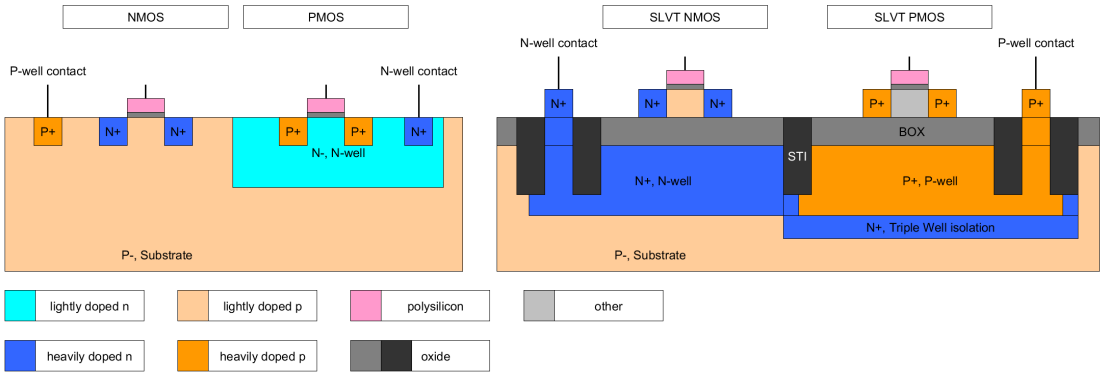


Figure 4.1: Left: CMOS cross section bulk, right: CMOS cross section FDSOI in flip well configuration

heavily doped N contacts create the source and drain within the weak p substrate of the NMOS transistor. The polysilicon gate is separated from the resulting p- channel with a thin oxide. To create a PMOS a weakly doped N-Well is created which is connected via a heavily doped N contact to create the bulk. Heavily doped P contacts within the N-well create the PMOS.

Indifference on the right side an FDSOI cross-section. Here the transistor is formed on a thin fully-depleted semiconductor on top of a buried oxide (BOX). Below the buried oxide a second substrate exists. Here two configurations are possible in this figure a forward body bias FBB also called flip-well configuration is shown. In this configuration, an N-well is below the NMOS transistor while a P-well below the PMOS. With this LVT (low V_{th}) and SLVT (super low V_{th}) transistors can be realised. The N-well below the NMOS creates the back bias and the P-well below the PMOS its corresponding back-bias. Both back-bias can be set to an individual potential. To prevent shorting the P-well with the substrate an N+ triple well, which is connected to the N-Well potential is used. The substrate can then be connected to its potential with a P+ contact. This connection is not shown in the above figure.

With the buried oxide FDSOI has several advantages in terms of lower leakage and being less error-prone to latch-ups since fewer NP transitions exist. Furthermore, the fully depleted channel achieves higher mobility and often is a different material for the PMOS to achieve better PMOS NMOS matching. Another advantage of no dopant in the channel is less sensitivity to layout dependent effects as the short channel effect or the length of diffusion effects and lower V_{th} variation due to random dopant fluctuations [36].

The additional back bias in FDSOI functions as a back gate effectively controlling V_{th} of the transistor, with it allowing to increase the frequency of circuit blocks or decrease leakage. Depending on the chosen configuration, flip well as shown in the above figure or conventional well where the NMOS is above a P-well. In this configuration different back-bias voltages can be applied according to the resulting diodes from the PN transitions. This results in the in FDSOI commonly used 5 and 6 terminal transistor devices. These devices are shown below with their symbol and the internal resulting diode connections from the PN transitions shown above. In this configuration, a back-bias voltage for the NMOS from $-V_{bias_n}$ to 0 and for the PMOS back gate from 0 to V_{bias_p} is possible without shorts. This allows the flip well configuration to lower V_{th}. The possible back-bias voltages would be switched in the conventional well configuration allowing increased

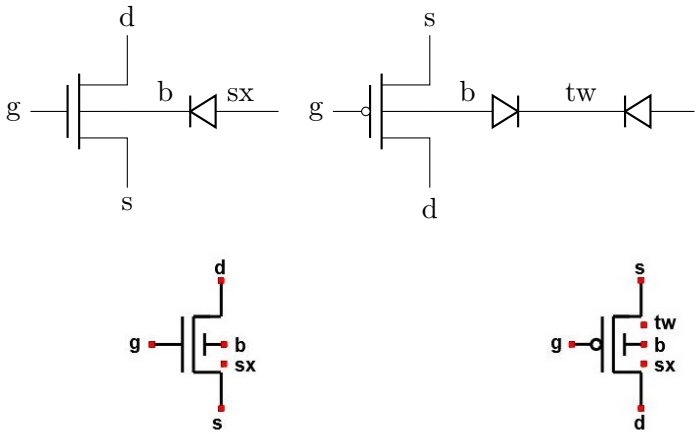


Figure 4.2: 5 and 6 terminal devices and corresponding diodes

V_{th}.

4.2 Electromigration

Electromigration (EM) has become a more prominent issue in smaller nodes. On the one hand, having a stack-up with a lot of thinner metals has more routing options for the router in Semi-Custom designs but on the other, it causes higher current densities and higher resistances. In figure 6.5 this development can be seen. The figure is from [37] where additionally a brief overview of the history and the challenges for EMIR are summarized. Another good resource for an overview is [38] which also summarizes most literature used in this section.

Over the last years, some aspects of EM in the nanometer-scale were discovered which makes the relations more complex but become important for smaller shapes. One of the challenges in EM analysis of complex designs is that it is done with an extracted design view which only offers limited information and additional the em deck is a simplified version of the complex physical behavior.

This section gives a broad theoretical background for EMIR on a nanometer scale. This will be useful to fully understand some of the rules in the vendor EM rule deck and give away on how to interpret and handle them.

Electromigration is one part of the general material migrations which describe processes

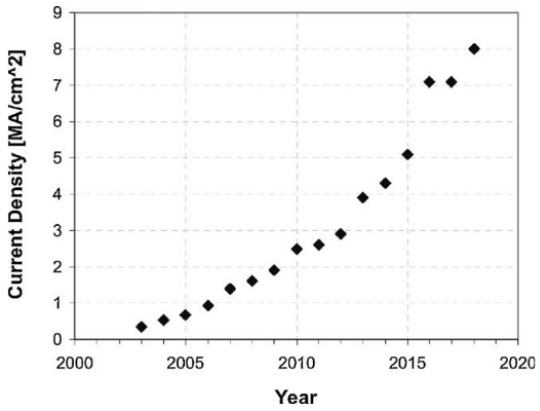


Figure 4.3: Current density trend from [37]

where material is transported in solid bodies. The current flow in a conductor produces forces to which the metal ions are exposed. In the figure 4.4 the forces are visualized. E_{field} is the electrostatic force. Which can be ignored most of the time since it is small compared to F_{wind} which is generated by the momentum transfer between electrons and the metal lattice. If this Force exceeds the activation Energy E_a a directed diffusion process starts and material transport takes place. From this several defects can arise either voids which will lead to opens and higher resistances or shorts caused by whiskers.

An empirical model which can be used to determine the mean time of failure was introduced by J.R. Black [39] in the 1960s as follows:

$$MTF = \frac{A}{j^2} * e^{\frac{E_a}{kT}} \tag{4.1}$$

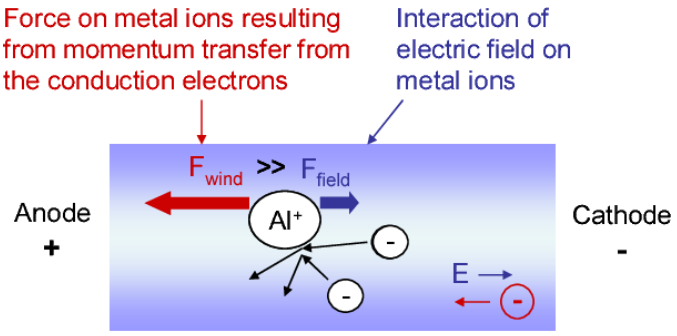


Figure 4.4: Forces on metal ions from [38]

Diffusion process	Activation Energy in eV	
	Aluminium	Copper
Bulk	1.2	2.3
Grain-boundary	0.7	1.2
Surface	0.8	0.8

Table 4.1: Activation Energy in Copper and Aluminium for different processes from [38]

Where A is a material constant including volume resistivity, a factor relating to the meantime of failure and mass transport, and many more. E_a is the activation energy as mentioned above, k the Bolzman constant, T the temperature, and j the current density. Later the exponent of the current density was replaced with n to model different failure modes in different metals (copper and aluminum).

With this, a relation between lifetime and temperature is given. There are more complex models additionally incorporating mechanical and thermal stress as in [40]. Another advantage is that their model does not rely on empirical values in contrast to E_a and A in Blacks Equation.

From Blacks Equation can be derived that one of the main property for EM is the activation energy E_a of a conductor. Which depends on the material and the location of the crystal lattice. Furthermore, the interconnect surroundings not only affect the lattice and with it E_a but also stress migration induced through mechanical stress.

For both of these aspects, effects are short-length effects and are becoming increasingly important for Nanometer Design. On the one hand, the grain size to wire geometry ratio and with it the Bamboo Effect and with mechanical stress canceling out electromigration the Immortal Wire Effect.

In table 4.1 the different activation energies for different diffusion paths and mechanisms are listed. The highest activation Energy needs bulk or volume diffusion. This mechanism is usually dominated by surface diffusion. There are some techniques for aluminum and some advanced processes for copper where the surface needs higher activation energy. At this point the grain boundary diffusion mechanism becomes prominent.

The most important aspect of the grain boundary diffusion process is the grain size. In amorphous structures, the crystal lattice is very dense and the grain boundaries are

so many that they are not relevant anymore. These structures can not be produced in metal interconnects. The other extreme is a monocrystalline structure where no grain boundaries exist. This condition is again not relevant yet since it cannot be reliably produced. The typical condition is a polycrystalline structure where the near bamboo and bamboo structures are polycrystalline structures with low grain density.

Bamboo Effect

The bamboo or near-bamboo effect occurs when grain boundaries are eliminated. This becomes the case for the two equally named structures mentioned above. The elimination of these paths acts positively towards EM robustness.

These structures can be grown by processes as tempering where the interconnect is backed at a high temperature and afterward slowly cooled as described in [41]

The main factor for the bamboo effect is the interconnect dimension especially the height and width of the wire. In Figure 4.6 from [42] the bamboo effect can be seen. In this work, the lifetime of different AL-0.5%Cu wires with varying line widths was observed. In this research, the grain size for the AL lines in their process was determined with an electron microscope. The grain sizes are between 1.2um and 7um with a median of 4u. It can be seen in the figure that at 2u the MTF reverses and the line becomes more robust. The paper states that below 2um a bamboo structure is generated where the grain boundaries are perpendicular to the current flow resulting in eliminating atomic flux.

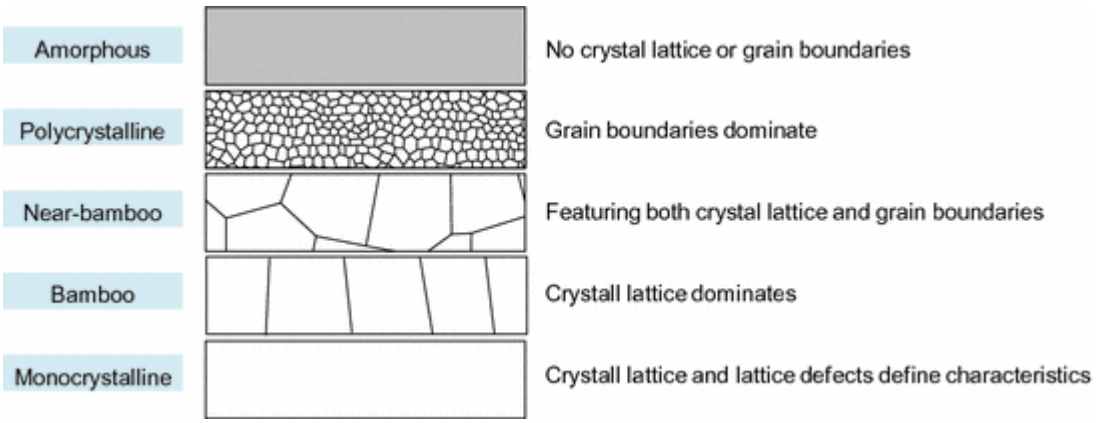


Figure 4.5: Different lattice structures from [38]

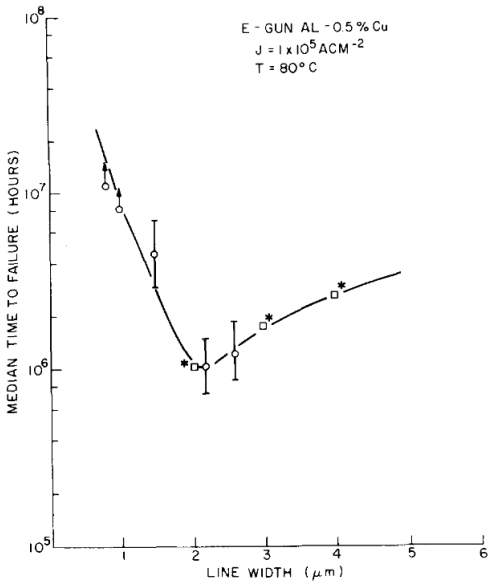


Figure 4.6: line width vs. MTF in AL-0.5%Cu [42]

The bamboo effect usually takes a role in tempered AL interconnects. This has become more important since aluminum interconnects are often used for the redistribution layer of a chip and becomes more critical since a high pin density has to be reached. In copper the surface diffusion dominates. Nevertheless, the surface diffusion in copper can be minimized to achieve an EM robust interconnect, and accordingly, the grain boundary diffusions and the bamboo effect becomes important again. Furthermore, in [43] it was found with specific annealing conditions and operating conditions for polycrystalline lines a mixture of the grain boundary and surface diffusion is dominant and for bamboo or near-bamboo structures solely surface diffusion.

Immortal Wire Effect

In 1975 I.A. Blech observed that no electromigration seems to take place in short aluminum interconnect stripes [44]. This is today called the blech effect where the critical length for which this effect holds is called blech length. The reason for this phenomenon is an equilibrium between stress migration and electromigration below the "Blech" length. Blech further discovered that the critical length depends on the current density and formulated the critical product $(jL)_{Blech}$. Below this critical product, no damage due to electromigration could be observed.

Vendor EM Rules

The rules used in a vendor PDK have to be simple since they are used for an increasing number of shapes during the electromigration check/analysis. Due to this in the PDK included EM rules are mapped as multiple linear functions with different offsets and slopes depending on the width and length. This leads to non-linear jumps in the maximum allowed current density function which is difficult to explain physically. It rather can be explained as a worst-case estimate by the vendor. Nevertheless, these estimates and simplified rules have to be used since they are the best data available.

5 Full Custom Schematic Generation

The schematic generation chapter will describe the implemented contributions of this thesis to increase the design efficiency in terms of circuit generation and sizing concerning consistency between all views. The implementation for this and the corresponding building blocks is given in the following paragraphs.

The ASIC design framework in use is the Cadence Virtuoso Framework, which is described in chapter 2. To implement solutions independent from proprietary languages like SKILL, a language bridge was developed. The skillbridge interface is a language mapping between python and SKILL. The development and concept of this language mapping will be described in this chapter.

In the Top-Down approach from [23] a hierarchy generation from the system-level description to schematic views was already in place. The Hierarchy Generation from [23] runs Cadence Genus with a synthesis script to elaborate the system-level design. This synthesis script is written in Tcl and creates black-boxes/-stubs from the system level leaf cells to transform it to synthesizable Verilog for Genus. The elaborated design is then used as input for Cadence VerilogIn [45] which can create the needed schematics views in the *oa database*.

Initially the scripts from [23] were used. It was quickly noticed that a lot of information about the elaborated design is needed to produce elaborated parameters for the leaf cell generation to ensure consistency between model leaf cell and implementation. Because of the third-party tools used in this script the script it was not possible to extend it.
In this thesis, an extendible version of the schematic generation was implemented to allow leaf cell generation and sizing. The implemented classes and flow offer a simple interface to the elaborated design. The new schematic generation will be detailed in the following.

An overview of this approach can be seen in figure 5.1. An important factor for the generation is the connection to the system level and the elaboration and resolution of

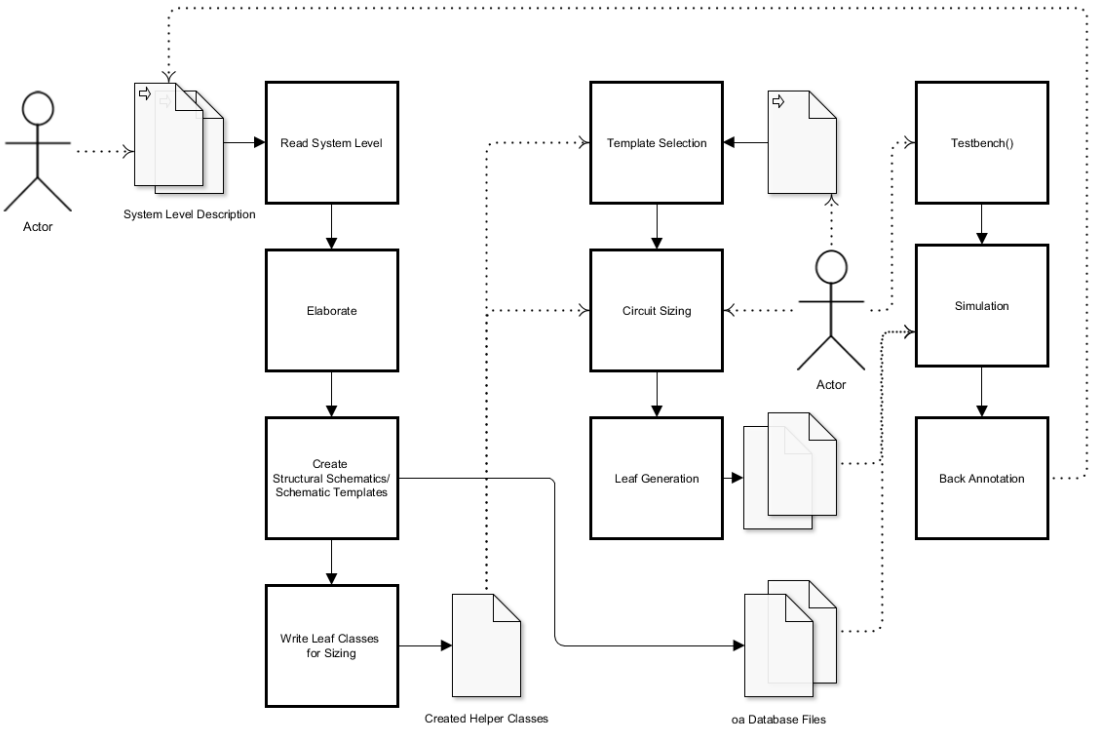


Figure 5.1: Flow Diagram Schematic Generation

generic parameters to a specific implementation. For this, the system-level description is read in and elaborated. Afterward, structural schematic views for Virtuoso and schematic templates are generated. In the last step config files with elaborated leaf parameters and ports are written which can be used to implement the leaf schematic generation.

The leaf cell generation is a very interactive approach where the circuit designer can create schematic templates which can be selected and sized with the sizing scripts. The focus for these sizing scripts is to provide an accessible way to implement expert knowledge in a technology-independent manner.
The parameters from the elaborated design and local parameters of the leaf cells are used to write config files which are used in the circuit sizing step. These sizing scripts will be executed to generate the leaf circuit. The leaf cell generation is detailed in this chapter in 5.3.
The advantage of this is it allows to create parameter-dependent sizing scripts which can be reused. For the sizing, a correspondence to the system level is ensured to keep

parameters consistent between all views for a specific design and a specific system-level parameter set.

To achieve a high accuracy between system-level simulation and implementation additional feedback is needed. The RNM parameters are often used as the specification for the leaf cell implementation. These specifications are transferred to the spice simulation of the implemented leaf cell. With the spice simulation, the specification is measured accurately and back annotated to the system level. Via this feedback an accurate system-level simulation is possible which can also be run over corners.
The leaf cell generation and back annotation are semi-automatic to offer as much reusability and usability as possible.

5.1 Skillbridge

The standard scripting language in the Cadence analog ASIC and PCB environment is SKILL. SKILL is a proprietary programming language based on Lisp.

SKILL as a proprietary language has several disadvantages to work with. The package ecosystem and user base of SKILL are small compared to other programming languages. While SKILL is a general-purpose programming language the proprietary nature it results in it being mostly used in the domain-specific fields of ASIC design (Virtuoso) and PCB design (Allegro). The use of SKILL in only these two very specific domains results in a small user base. The small user base results in additional problems as also fewer experts and only a single source of language documentation, application notes, and example code. Furthermore, it is difficult to find experienced SKILL engineers or programmers as Lisp is not a commonly used language either.

Modern development flows include continuous integration and deployment. Continuous integration will run code analysis, tests, and test coverage with every commit to the project. These cannot be realized with SKILL as it means each time a test or code analysis is executed a license needs to be checked out. Licenses are scarce and expensive making this approach not feasible. The goal of the skillbridge is to separate the control code from the SKILL interpreter.

Another problem resulting from the restricted usage domain and proprietary nature is the state of the SKILL package ecosystem which does not exist. This is further ex-

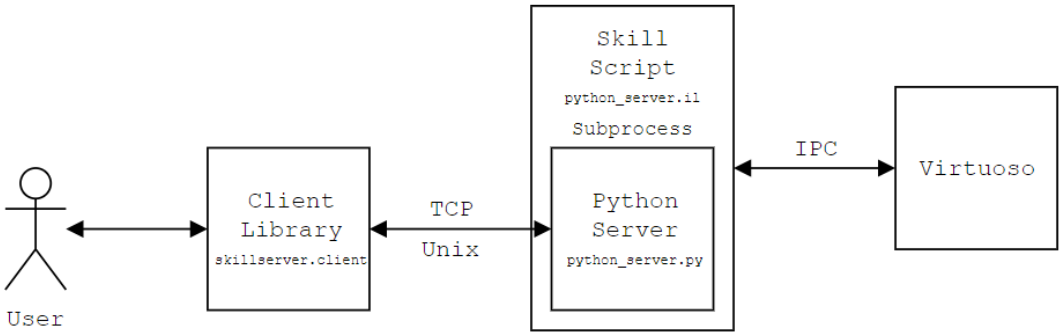


Figure 5.2: Communication components Skillbridge [50]

aggerated with SKILL not offering a package system and the fact that sharing code in the PCB design and ASIC domain is still rarely done. Reusable code snippets are from cadence, which can be accessed via a support account and are given as they are. There is also an effort from Robert Bosh Zentrum (RBZ) to introduce a package manager to SKILL (SPAM) [46] and offer some packages. While this is a good approach it is not widely adopted or used.

To overcome these restrictions and difficulties, a dynamic language mapping is introduced. It provides the user with a more widespread language, which simplifies the access and abstracts SKILL to this widespread language.
One of today's most popular and widely used programming languages is python. Its popularity has risen in the last years, popularity according to Tiobe index rank 3 [47], according to Github projects rank 2 [48] and according to the yearly StackOverflow survey rank 4 [49]. Pythons popularity results in a huge user base and package ecosystem. The package ecosystem is one of the main reasons python is is heavily used in various other domains besides computer science. Prominent examples are geology, biology, mathematics, and engineering disciplines. Especially the packages from the mathematics domain make python a perfect fit as a scripting programming language for circuit design.

The skillbridge was developed in cooperation with former research assistant Niels Buwen and is now maintained as an open-source project on [50]. It is an interface between Python and SKILL that achieves a seamless language mapping. In the following paragraphs, the skillbridge approach to communicate with Virtuoso is explained. Afterward, all translations to offer a seamless interface are detailed.

One main challenge which had to be solved is the communication with the SKILL interpreter inside Virtuoso since there is no interface to call SKILL functions from the outside. The basic concept to solve this issue is shown in figure 5.2. There is a SKILL script (python_server.il) running in the Virtuoso CIW which starts a python server (python_server.py) in a subprocess and establishes an inter-process communication (IPC) where the stdin and stdout from the CIW are redirected to and from the python server. The python server opens a Unix socket for a client script to connect to.

The client skillbridge library offers all the functions to call the global SKILL methods and translate types and objects. With this, seamless integration into python is achieved. The mapping is dynamically implemented with dunder methods. Double underscore (dunder) methods are called indirectly and define specific object behavior, like operator overloading, context management, or sequence behavior. With these methods, skill methods can be build and send to the interpreter without the need to explicitly map them by hand and powerful IDEs as PyCharm [51] or Jupyter Notebook [52] are supported by implementing other useful dunder methods as dunder dir.

Besides the communication approach shown above, there is also a direct mode where the client script communicates directly to the server without the socket communication and solely the inter-process communication. For dynamic use cases, socket communication is ideal while for fixed tooling the direct communication can be used.

5.1.1 SKILL $<->$ Python translation

To offer a seamless interface three main mappings have to be implemented. The object mapping of SKILL objects to python objects has to be done. With this, these objects can natively be used. Additional types have to be mapped. SKILL types have to be converted to python and python types to SKILL. Also, the global SKILL methods and with it, the interfaces to the Virtuoso subsystems have to be mapped.

The behavior shown below can be implemented with dunder methods which are special methods in python to realize or manage operator overloading, sequence access, or iterators. The most important implemented dunder methods are described below and are shorted from the actual implementation which can be found at [50]. The array behavior and slices are not implemented on SKILL side. They come automatically with the type translation described in the next paragraph.

```
1  # simple SKILL object attribute access
2  # cv->cellName
3  # should be in python:
4  cv.cell_name
5  # more complex access SKILL object member which is a list of objects
6  # (nthelem 10 cv.shapes)->bBox
7  # should be in python
8  cv.shapes[10].b_box
9  # additionally python slices and list comps should work
10 shapes_slice = cv.shapes[3:20:2]
11 m1_shapes = [shape for shape in cv.shapes if shape.layer == "M1"]
```

For basic access __get_attr__ and __set_attr__ are implemented as shown below.
For __get_attr__ the attribute goes to the translator, where it is converted to the corresponding SKILL attribute (line 3). Afterward, the result is decoded to the corresponding python type and returned (line 4).
The __set_attr__ method is used to set an attribute. Again the corresponding SKILL form is generated in line 8 and afterward, the result is converted to check if for possible error handling.

```
1 def __getattr__(self,key: str) -> Any:
2     ...
3     result = self._send(self._translate.encode_getattr(self._variable, key)
      )
4     return self._translate.decode(result)
5
6 def __setattr__(self, key: str, value: Any) -> None:
7     ...
8     result = self._send(self._translate.encode_setattr(self._variable, key,
       value))
9     self._translate.decode(result)
```

Another important dunder method implemented is __dir__ which contains a list of all mapped object attributes. The dir method in python calls this dunder method. It is used in example in Jupyter Notebook for tab completion.

```
def __dir__(self) -> List[str]:
    response = self._send(self._translate.encode_dir(self._variable))
    attributes = self._translate.decode_dir(response)
    return attributes
```

In the following section, some examples of types that have to be mapped are introduced. Each type coming as a result from SKILL has to be mapped to python and each type given to SKILL to a SKILL type.
In terms of float, int, or string no converting has to be done. Here both languages are compatible. Some differences are shown below.

```
# SKILL boolean {t, nil}
# in python
{true, false}
# SKILL list '(1 2 3 4)
# in python
[1, 2, 3, 4]
# Sometimes SKILL uses symbols i.e. (simulator('spectre))
# in python
ws['simulator'](Symbol('spectre'))
```

The translation from python to SKILL and the other way around is implemented as string conversions and manipulation. For the translation from python to SKILL, this is done in the python skillbridge client library. For the return values, the same mechanism is implemented on the SKILL server side. This combination of only converting types to strings and not having to parse strings offers the best performance.

The last part of the translation is the global function mapping. In lines 1-3 the most simple function call is shown. The difference on the Python side is, that we access the functions over a RemoteFunctionGroup Object i.e. in line 3 ws.ge. With this approach, we group SKILL methods by prefixes. Skill interfaces to different subsystems in virtuoso have mostly a corresponding prefix. Additionally, it is possible to access the function with snake-case. Snake case is more common within python projects and this will result in a more unified and clean codebase. Furthermore again all types which are given to the function should be translated automatically as shown in line 4-7.

```
# SKILL method call geGetEditCellView()
# in python
ws.ge.get_edit_cell_view()
# SKILL method with attributes
# dbOpenCellViewByType(lib->name, cv->name, "schematic", "", "a")
# in python
ws.db.open_cell_view_by_type(lib.name, cv.name, "schematic", "", "a")
```

Again a dynamic approach was chosen where if a Remote Function object is called the corresponding __call__ is triggered. This method can be seen in the code snippet below in line 4. At first, the RemoteFunction method lazy is called to decode the command which has to be sent to via the _channel.send method. The result from the function is then translated to a python type and returned. The lazy method takes the arguments and keyword arguments given from __call__ the dynamically generated function name is read out and transformed to camel case with all the information needed the call gets encoded to valid SKILL code and the resulting command is returned.

```
...
class RemoteFunction:
  ...
    def __call__(self, *args: Skill, **kwargs: Skill) -> Skill:
        command = self.lazy(*args, **kwargs)
        result = self._channel.send(command)

        return self._translate.decode(result)

    def lazy(self, *args: Skill, **kwargs: Skill) -> SkillCode:
        name = snake_to_camel(self._function)
        return self._translate.encode_call(name, *args, **kwargs)

    def __repr__(self) -> str:
        command = self._translate.encode_help(self._function)
        result = self._channel.send(command)
        return self._translate.decode_help(result)
    ...
```

The most important feature of the skillbridge is this dynamic behavior. None of the RemoteFunction objects or RemoteFunctionsGroup objects are directly mapped to SKILL

prefixes or methods. The dynamic mapping of these results in a future proof approach where if SKILL interfaces are added or changed to Virtuoso they are automatically mapped accordingly.
The mapping is implemented dynamically with a handful dunder methods. In the below example it is explicitly written which internal dunder methods are executed when the user accesses a global method. In line 2 the user method call is given. In lines 4-6 the resulting actual call is shown. *ws* is a *Workspace* object which implements *__get_attr__* and calls it on attribute access with the attribute key as parameter. *Workspace.__get_attr__(key: str)* returns a *RemoteFunctionGroup* object which implements its own *__get_attr__*. This is called with the function key as the parameter and returns a *RemoteFunctionObject*. *RemoteFunctionObject* finally calls the above described *__call__* method which is triggered on call and passes the call's parameters.

```
1 # how the user accesses the method (dunder methods are called indirectly)
2 ws.db.open_cell_view_by_type(lib.name, cv.name, "schematic", "", "a")
3 # how it is actually called via the implemented dunder methods
4 ws.__get_attr__('db').__get__attr__('open_cell_view_by_type').__call__(
5   lib.name, cv.name, "schematic", "", "a"
6 )
```

5.1.2 Usage Examples

A minimal example for a client application. The examples here can also be found in [50]:

```
1  from skillbridge import Workspace
2
3  ws = Workspace.open()
4  ws["load"]("userFunctions.il")
5  cellview = ws.ge.get_edit_cellview()
6  cellview.b_box # returns [[0, 10], [2, 8]]
```

In line 1 the Workspace class of the client library is imported. The workspace corresponds with Virtuoso's Command Interpreter Window (CIW) and can be connected with the server with the open command, line 3. When starting the server a specific port can be selected. This port is also an optional parameter for open to connect to the corresponding Virtuoso instance. This allows multiple server/ client pairs on one host.

With the workspace either a function of a specific Virtuoso subsystem SKILL interface can be called or a simple SKILL call.

Normal SKILL calls or user-defined functions can be accessed as shown in line 4 where the SKILL load function is called. The function call is given in square brackets and the arguments for the function in normal brackets, as python datatypes.

The SKILL methods of a specific Virtuoso subsystem normally have a common prefix. The prefix is used in the skillbridge to subgroup all available methods. In these examples geGetEditCellView from the ge (graphical editing interface). Which corresponds to ge.get_edit_cellview() as seen in line 5. The grouping by prefix also allows a better tab completion where not every method is matched but every method corresponding to the prefix category.

The method returns a cellview object where attributes can be accessed as shown in line 6. Every return value is translated to a python data type from SKILL.

A more complex application is shown in the example below where instance statistics of a specific layout are read out and plotted in a pie chart. As in the example before the Workspace is opened. Afterward, we get the current cellview line 5-6. A helper function to calculate the area of an instance from its bounding box is defined bbox_to_area. In line 12 a list comprehension is used to create a list of cell names with the corresponding area from all instances in the cellview cv. Afterward, with the help of Counter a dictionary for the instances and the instance occurrences is generated. Furthermore, a dictionary for the instances and their area is created lines 15 - 22. From lines 25 to 30 the plots are prepared and the number of instances and their accumulated area is plotted in a pie chart.

```
1  from skillbridge import Workspace
2  from collections import Counter
3  from matplotlib.pyplot import pie,figure,title
4
5  ws = Workspace.open()
6  cv = ws.ge.get_edit_cell_view()
7
8  def box_to_area(b_box):
9      return (b_box[1][0]- b_box[0][0]) * (b_box[1][1] - b_box[0][1])
10
11 # Get a tuple of instance cellname and area
12 insts = [(inst.cell_name,box_to_area(inst.b_box)) for inst in cv.instances]
13
14 # Get a dictionary of cell_name and occurences
15 counts = Counter(name for name, _ in insts)
16
17 # create dictionary of cell_name and area
18 areas = {}
19
20 for name,area in insts:
21     areas.setdefault(name,0)
22     areas[name] += area
23
24 # plot the pie chart
25 f = figure(figsize=(12, 12))
26 sub1 = f.add_subplot(121)
27 sub1.set_title("Number of instances")
28 pie(counts.values(), labels = counts.keys())
29 sub2.set_title("Accumulated Area of each Cell")
30 pie(areas.values(), labels = areas.keys())
```

As shown in these examples the mapping is seamless. Natural Python constructs can be used as shown in the examples with i.e. list comprehension or array access. The user does not have to think about the SKILL code in the background and can concentrate solely on programming in python.

Performance-wise a limiting factor can arise when a lot of SKILL objects have to be accessed. In the below example the normal python for loop lines 4-6 can be slow when the objects in cv.shapes are very large. In this case, we filter for layer "M1" and delete these shapes. Most of the access behavior is implemented with Python's dunder methods

iterating and filtering large Remote object list will result in many accesses to the CIW. This throughput to the CIW is limited by Virtuoso and thus limits performance in these cases.
In the skillbridge, there are several helper functions implemented to mitigate these performance issues. One of these helper methods is the lazy attribute. This attribute marks Remote objects to be evaluated later. With this, a filter can be constructed as seen in line 15 which then will be sent with one access to the CIW and executed on SKILL side.

```
1  from skillbridge import Workspace
2  # Pythonic approach may be slow for very large lists
3  cv = ws.ge.get_edit_cell_view()
4
5  for shape in cv.shapes:
6      if shape.layer == 'M1':
7          ws.db.delete_object(shape)
8
9  # With lazy evaluation List needing only one CIW access
10
11
12 ws = Workspace.open()
13 cv = ws.ge.get_edit_cell_view()
14
15 shapes = cv.lazy.shapes
16 shapes.filter(layer='M1').foreach(ws.db.delete_object)
```

If the task does not include mainly accessing SKILL objects but instead has a huge processing part. Such an application using Python and the skillbridge will most likely improve performance since most numerical or mathematical libraries in python use hardware acceleration.

5.2 Hierarchy Generation

In this section of this thesis the hierarchy generation will be described which is needed to convert a system-level model to a specific implementation.

Initially, a reference design will be introduced on which the implementations are done in this chapter will be explained.

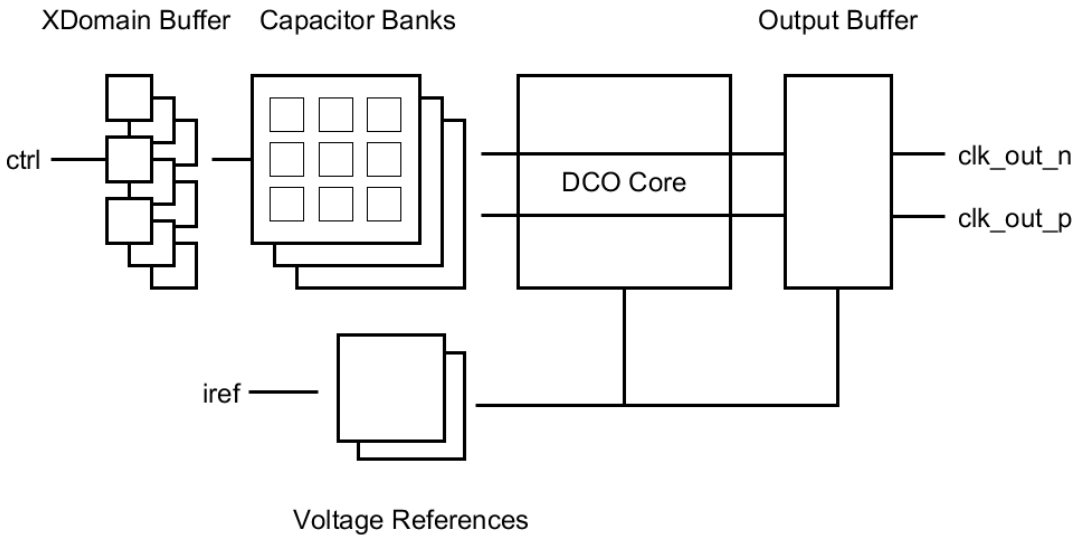


Figure 5.3: Simplified structure reference DCO design

Afterward, the general design organization will be discussed. This is an important topic since a lot of different views exist in the design and multiple implementations could be needed for one system level. The file structure and repository organization is important as it is a good foundation for later reuse.

The system-level description is the input for the schematic generation. From this input, an internal design representation will be created.
On the internal design representation, further tools can be implemented. One of them is the generation of the structural schematics in a *oa database* for the Virtuoso Environment.

The implementation and object hierarchy will be discussed in detail in the following subsections.

5.2.1 Reference Design

A Digital Controlled Oscillator (DCO) design is introduced as a reference in this chapter. With this, each processing step can be explained in a simple example. The DCO is a typical Full-Custom block in larger Mixed-Signal designs, often the only Full-Custom block inside an All/ Mostly Digital PLL.

The DCOs hierarchy is shown in figure 5.3. The oscillator block is the DCO core which will model the LC tank and amplification required for oscillation. The internal oscillating node connects to several capacitor banks, shown on the left. The coarse bank is needed to tune out the process corner or to cover the needed frequency range. Finer capacitor banks are used by the loop filter when integrated into a Phase Lock Loop (PLL) to achieve a controlled frequency output. Additionally, cross-domain buffers are used for the control bits of the capacitor banks to offer a separation between the digital and the analog domain. Furthermore, an output buffer for the frequency output exists to drive higher loads and decouple the oscillation node as well as possible. Some voltage reference generators are needed for the output buffer and the DCO core.

The reference design's system level is implemented in a Top-Down manner according to [23]. The analog behavior is modeled with System Verilog as Real Number Models (RNM).

For our RNM models, two aspects are important. The system-level must be fast to simulate complex system-level test cases. Implementing a system-level model with only the important parameters and a high abstraction increases simulation performance significantly. The second important modeling approach is to model each behavior in the corresponding leaf cell description in our case the RNM view. This is especially important to later back annotate block-level simulations to the corresponding block.

In our DCO only two main factors are important to simulate. These are the correct output frequency behavior and the jitter. At the system-level, the abstraction is chosen in a way to only model these two important parameters. Modeling the jitter and frequency behavior is important to verify the behavior of other blocks affected by the DCO for example a PLL using the DCO or other modules, the output jitter of a clock generator is one of the most significant factors impacting overall performance.

Thus the frequency generation is done in the RNM view of the DCO_CORE. The oscillation frequency is calculated based on local parameters and the oscillation nodes. The local parameters represent the inductance and capacitance of the LC tank for the oscillator, while in the oscillation node the summarized capacitance of this node is represented. A capacitance change in the capacity bit should result in a frequency change in the DCO_CORE. This is modeled via a node capacitance which is implemented with

user-defined nettypes and user-defined resolution functions. These are available in System Verilog.

```
1  typedef struct {
2    logic net = 1'bZ;
3    real C;
4  } wire_cap;
5
6  function automatic wire_cap capSum (input wire_cap driver[]);
7    bit MULT_DRIVERS_FLAG = 1'b0;
8    foreach(driver[i]) begin
9      capSum.C += driver[i].C
10     if (driver[i].net === 1'bZ) begin
11       capSum.net |= 1'b0;
12     end
13     else begin
14       capSum.net = driver[i].net;
15     end
16   end
17 endfunction
18
19 nettype wire_cap wCapSum with capSum
```

Both are shown in the above code listing. A custom nettype can be implemented as a struct (line 1-4) and thus build-up from multiple values. In our example a real value for the capacitance of the net and a logic value for the oscillating node. Additionally, an implemented resolution function will be called when multiple nets are connected (lines 6-17). When multiple drivers exist the capacitance is summed up. The net will also be evaluated based on the drivers where only one driver is allowed and all other drivers have to set the net to high impedance 'Z'. The nettype definition is shown on line 19. The behavior of these nettypes and resolution functions are also described in [23] for a ring oscillator circuit based on delay cells.

Another important feature is the auto-coercion of port type declarations. The leaf cells can have different views. Thus the type of the ports can vary while in the RNM view a net might be a custom nettype as our defined wCapSum the same net in a function view might be logic and in a more specific VerilogAMS model might be electrical. Auto-coercing allows this type to be inherited through the hierarchy even when only wires are defined and thus allowing the same structural design descriptions for all leaf cells. In the event-based simulator XCellium the types wreal, SystemVerilog wreal nettypes, and

SystemVerilog user-defined nettypes are supported for coercion. Additional information about coercion and user-defined nettypes can be found in chapter 9 (Real Number Modeling) in [53] and in [54] under 6.6.7 User-defined nettypes.

An additional local parameter is the intrinsic jitter of the dco_core which additionally manipulates the output frequency.

The oscillator is implemented as an event generator based on the inductance and capacitance parameter and an additional random component, modeling the jitter and changing the time each event is created. This results in a very fast overall model since the number of events generated mostly influences the run time. In this approach, the generated number of events is independent of the added jitter.
In the listing below the event generation in the DCO_CORE RNM model is shown. In the first initial block lines 1-7 some pre calculations are done. From the local parameters, *VCO_TANK_L* and *vco_tank_c_fixed* and the capacitance component of the node the ideal period and oscillation frequency is calculated.

```
1  initial begin
2    K = $sqrt(2);
3    total_jitter = $sqrt(RMS_INTRINSIC_JITTER**2 + RMS_PXF_JITTER**2)/K;
4    vco_c = CLKOUT_P.C + vco_tank_c_fixed;
5    ideal_period = 2.0*`M_PI*$sqrt(VCO_TANK_L*vco_c);
6    ideal_freq = 1.0/ideal_period;
7  end
8
9  initial begin
10   startup_delay = rdist_normal((20e-12/`TSCALE), (20e-12/`TSCALE),100,0);
11   #(startup_delay);
12   forever begin
13     dT = ((ideal_period/2.0) +
14     (total_jitter/1000)*$dist_normal(seed, 0, 1000)/`TSCALE;
15     #(dT) CLKOUT_P_int = ~CLKOUT_P_int;
16   end
17 end
```

The startup delay is implemented in the following initial block. The event generation for the oscillation is implemented within a forever loop. The oscillation node is repeatedly inverted with a frequency of half the period plus a random distribution which is

multiplied by our calculated jitter and scaled to the correct time scale.

Another aspect of the modeled DCO is that each capacity bit must influence the oscillating frequency. For all different capacitor banks, a general RNM model named MGT_MODEL_CAP_BIT is used. This model is shown in the listing below. Depending on DIN the capacitance transmitted over the wCapSum net is increased by a parametrized switching capacitance (CAP). An offset capacitance (C_OFF) is always added which is the second parameter of this module.

```
1  module MGT_MODEL_CAP_BIT #(
2    parameter real CAP = 1e-15,
3    parameter real C_OFF = 0.1e-15
4  )(
5    input wire VDD,
6    input wire VSS,
7    input wire DIN,
8    inout wCapSum INN,
9    inout wCapSum INP
10 );
11   assign INN = wire_cap'{1'bZ, C_OFF + DIN* CAP};
12   assign INP = wire_cap'{1'bZ, C_OFF + DIN* CAP};
13 endmodule
```

Additionally, the MGT_MODEL_OSC_BUF also has a parameter to model its input capacity.

5.2.2 Design Organisation

Large Hardware Designs can be compared with large software projects where multiple parts are separated into several repositories to increase reusability. While in this case hardware and software projects can be compared there are some important differences. In software development, a module has exactly one view with a single corresponding File. In a complex Top-Down hardware design, multiple views of a module exist. For a System Level model, the following defined views are important to know.

Definition 1. Non leaf cells should only contain a structural description of the design and are categorized as *structural view*

Definition 2. Synthesizable leaf cells are categorized as *digital view*

Definition 3. A leaf cell implemented as RNM is categorized as *rnm view*

Definition 4. A leaf cell containing a general RNM model is defined as *leaf view*

Definition 5. A leaf cell has a *functional view* which is a simplified model without any analog signal behavior for faster simulation.

For our reference design the file structure is shown in figure 5.4. The minimal DCO is in our case the parent directory. Each view in our design is represented as a folder. Furthermore, the file list MIN_DCO.f is given. The .f file is commonly used as input for event-based simulators or synthesis. For a design, multiple .f files can exist which allow selecting different views of the design if available (functional, RNM, AMS). In this example, the file structure is simplified. Normally the model files beginning with MGT_MODEL* are in a separated repository as they are often used and for high reusability separated in a model repository for general use in many different designs. Furthermore larger designs can contain subdesigns organized in separate repositories with their structure and .f files.

In the file tree in figure 5.5 a typical file structure of an *oa database* for Virtuoso is given. Each implementation of a full-custom design part is represented by one *oa database* which should reside inside its repository. The system-level files are given via symbolic links to the repository containing the system-level description of the design. It is important to separate both designs since the system level is independent in contrast to the implemented technology. From the technology-independent system-level various implementations for different technologies or stack-ups then can be derived. Technology dependent views in the implementation are:

Definition 6. A *schematic view* contains the netlist information of the cell.

Definition 7. A *layout view* contains the layout geometry.

Additionally the implementation includes technology-dependent testbenches for each module to test its specification. Here schematic views are used to create a test deck for the device under test.

Definition 8. A *maestro view* containing technology corner, simulator setup and optional informations for sweeps, post processing and specs.

Definition 9. A *config view* is used to define which views should be included for each module for the simulation.

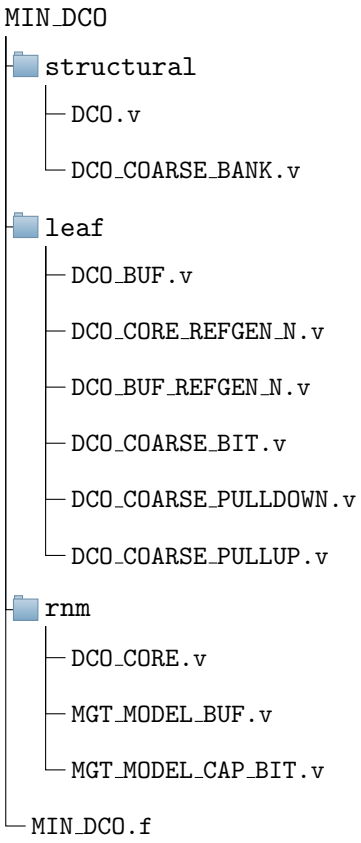


Figure 5.4: File organisation of the system-level description

5.2.3 Structure Generation

The described design with its organized file structure and different repositories is the input for the schematic generation. In this subsection, each step needed to create the structural schematics is described and how they are implemented. First, the design is elaborated then the internal design representation is build up. Lastly, the actual schematic leaf generation takes place.

Elaboration

In the elaboration step, the system-level design will be parsed. Generate constructs and parameters will be resolved to result in a specific design for the given parameter set.

To achieve this the file list has to be read in. Each file has to be parsed with a defined top module. The class to read in .f files can be seen in figure 5.6. This class will

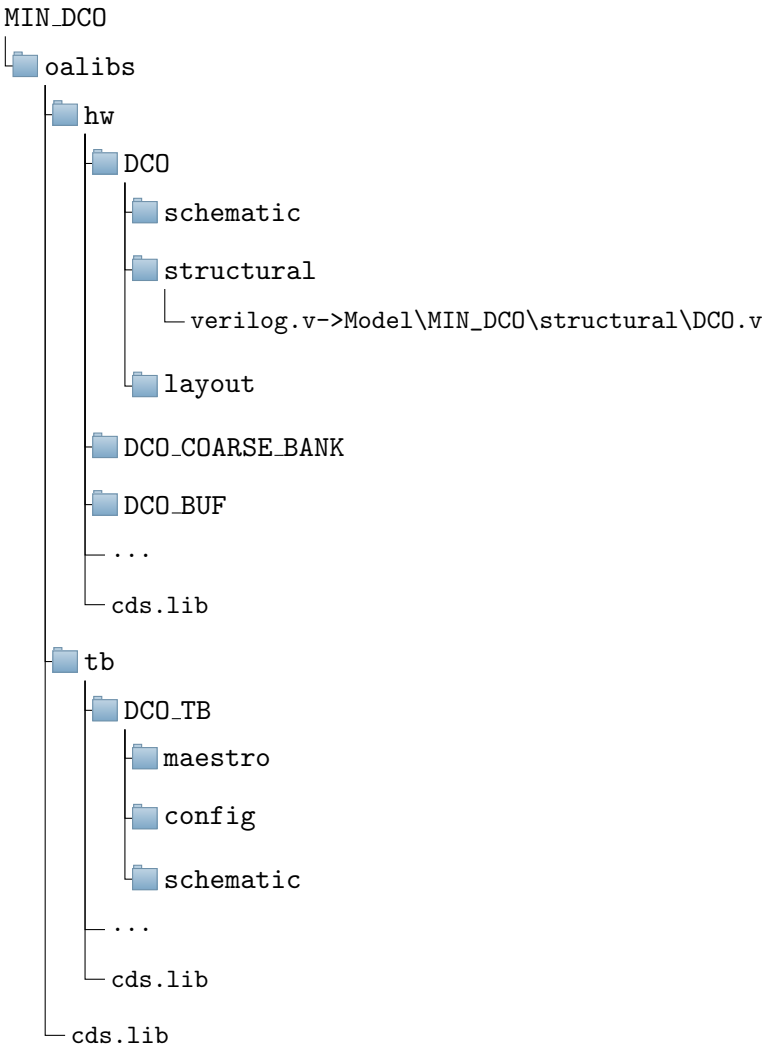


Figure 5.5: File organisation oa database

read in a .f file and supports hierarchical descriptions. The design files will be sorted into different views based on the folder structure.

In a first approach Cadence Genus [55] was used as a synthesis tool for parsing and elaboration. Tcl is used in Genus as a scripting interface. Via this scripting interface, the System-Verilog design can be parsed and elaborated. At the computer architecture group (CAG) a SystemVerilog parser was implemented with ANTLR in a student work [56]. With the help of this grammar, a little tool was implemented which transforms an elaborated Verilog design output by Genus to a JSON representation.

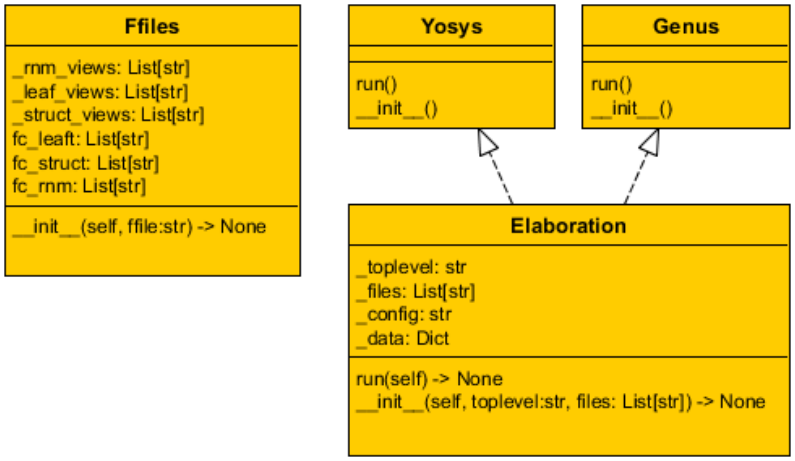


Figure 5.6: Classes implemented for the Elaboration

Performing the parsing and elaboration step with Genus has some disadvantages. Genus is a commercial tool for the synthesis of complex designs resulting in long start-up times and license checkouts. Additionally, two binary tools have to be called to parse, elaborate, and create the JSON representation.

An alternative approach was implemented with Yosys. Yosys is a lightweight open-source synthesis tool created by Clifford Wolf [57]. It thus does not need a license checkout. Yosys also offers a scripting interface either with its command language or a Tcl interface. Furthermore, Yosys offers a write_json command to write the elaborated design to a JSON file.

To run the parsing and elaboration an abstract interface class is implemented. This class allows us to interchange different approaches in our example the specific implementations Genus or Yosys to achieve tool independence. The resulting classes can be seen in figure 5.6. To implement a specific tool the abstract class methods *run* and *clean* from the general Elaboration class have to be implemented. The run method executes the Elaboration tool and writes the JSON file, while the clean method cleans up all files generated by the Synthesis tool for example all logs or unnecessary temporary files.

In the below listing the .f file is read in and the elaboration is executed with Yosys with the above-described object hierarchy. In lines 1 and 2 an include directory for the elaboration is set and the files are prepared with the Ffiles class. Since synthesis

tools cannot elaborate non-synthesizable code in the next step stubs for the leaf files are generated. The resulting stub file only includes the black-box description of the Verilog module and is thus treated as a black-box module for the synthesis tool. Furthermore for the stub generation not-synthesis supported nettypes and parameter types must be exchanged. This is done with the *type_changes* dict in line 3. Afterward, Yosys is called with the name of the top-level module, the list of design files, the include directory, and the name for the output JSON file.

```
1 include_dir = "MIN_DCO/"
2 files = Ffiles("MIN_DCO/MIN_DCO.f")
3 type_changes = {"wCapSum": "wire", "wreal": "wire", "string": ""}
4 stub_files = [create_stub(file, type_changes) for file in files.fc_rnm]
5 yosys = Yosys(
6       'DCO',
7       stub_files + files.fc_struct + files.fc_leaf,
8       include_dir,
9       'out'
10    )
11 yosys.run()
```

This summarizes the two approaches to elaborate the design. The Yosys approach allows us to work without commercial licenses.

Internal Design Representation

The goal for the internal design representation is to offer a defined interface to all attributes and methods needed for schematic generation from the elaborated design.

With the top module name and the JSON configuration, the internal design hierarchy is build up from a Module object. The class hierarchy in use can be seen in figure 5.7.
The Module is composed of a list of ports, nets, and instances. For each of these lists which are build up in the __init__ method a corresponding private method exists. The corresponding part of the read in JSON configuration dict is through these methods and the lists are built. The classes used to compose the lists of objects inside the Module class are:

The Net class offers three attributes, a name, the number of bits in the net, and a

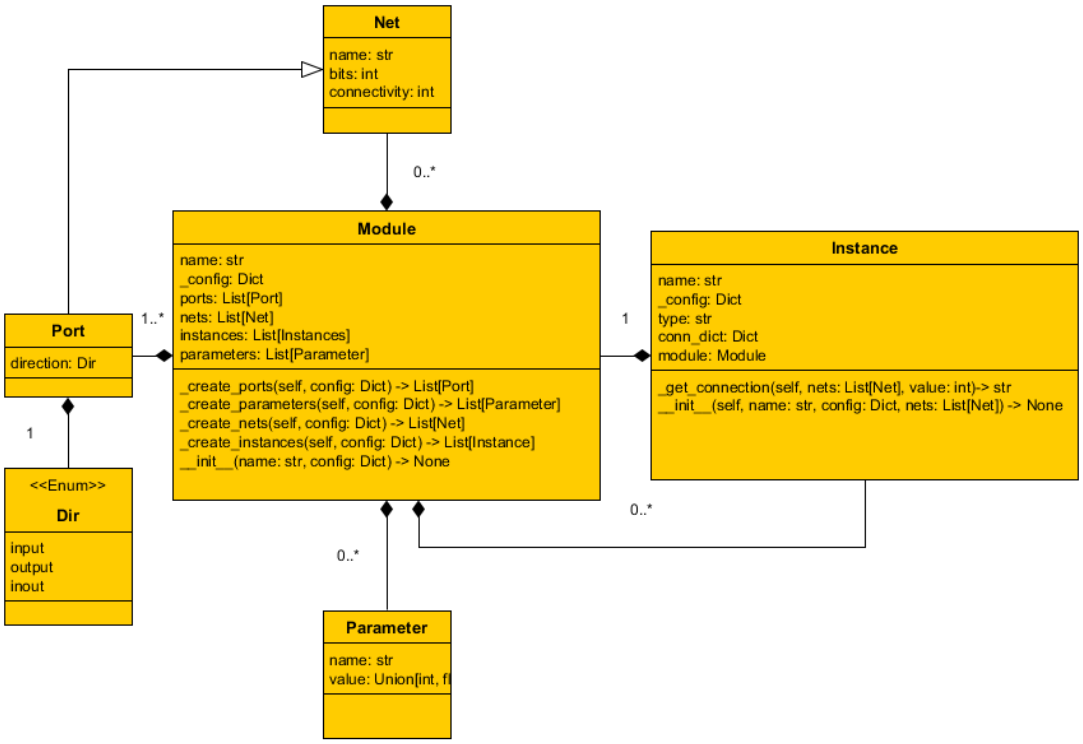


Figure 5.7: Class hierarchy internal design representation

connectivity identifier. A Net object represents internal nets from an elaborated Verilog module. The Port Class inherits these attributes and adds a direction which is an enumeration class offering the selections input, output, and inout. A Port has exactly one direction and represents a port of a Verilog modules port declaration.

The *Instance* class corresponds to a Verilog instance declaration. The attributes of this class are an instance name, the corresponding config dictionary from the JSON file, and a connection dictionary that holds information on how the ports are connected in the module. A *Module* holds a list of instances where the corresponding *Instance* is defined. An *Instance* class is composed of exactly one *Module*. With this, the design tree is span since these modules may again include a list of instances.

Schematic View Generation

Structural schematics can be produced with the internal design representation.

Similar to the *Elaboration* class an abstract interface *SchematicEntry* exists to implement the generation independently from the Virtuoso SKILL interface. A skillbridge workspace is open from the implementation *VirtuosoSchematicEntry*. In figure 5.8 the class hierarchy and their methods can be seen. The *SchematicEntry* class offers a tool-independent interface.

The abstract methods of the abstract class have to be implemented by the specific implementation class in this case by *VirtuosoSchematicEntry*. The defined tool independent methods are schematic manipulation commands as *create_view* which creates the cell view, *create_pins*, and *create_instances* which take elements or lists of the above-described design description structure. The function *create_symbol* is used to create a symbol from a cellview and its drawn ports. conn_by_name takes an Instances, searches it in the schematic, and uses the connection dictionary from the Instance object to create the corresponding connections for the instances to create connect-by-name wires and labels.

5.3 Leaf Cells Generation

For the leaf cell generation the flow changes to a semi-automatic incorporating the circuit designer. The reason for this is that an automatic generation approach for the generation of the leaf cells has several disadvantages. For one decision have to be made at the system-level which topology is chosen and which sizing script is run. This either resulted in a lot of additional parameters for the RNM leaf cells which were only used as hints for the generation or in an often unwanted behavior that was difficult to direct.

A semi-automatic approach does not suffer these problems and is automatically com-

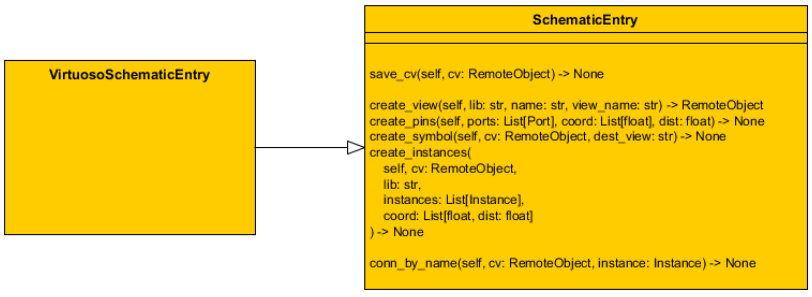


Figure 5.8: SchematicEntry and Implementation

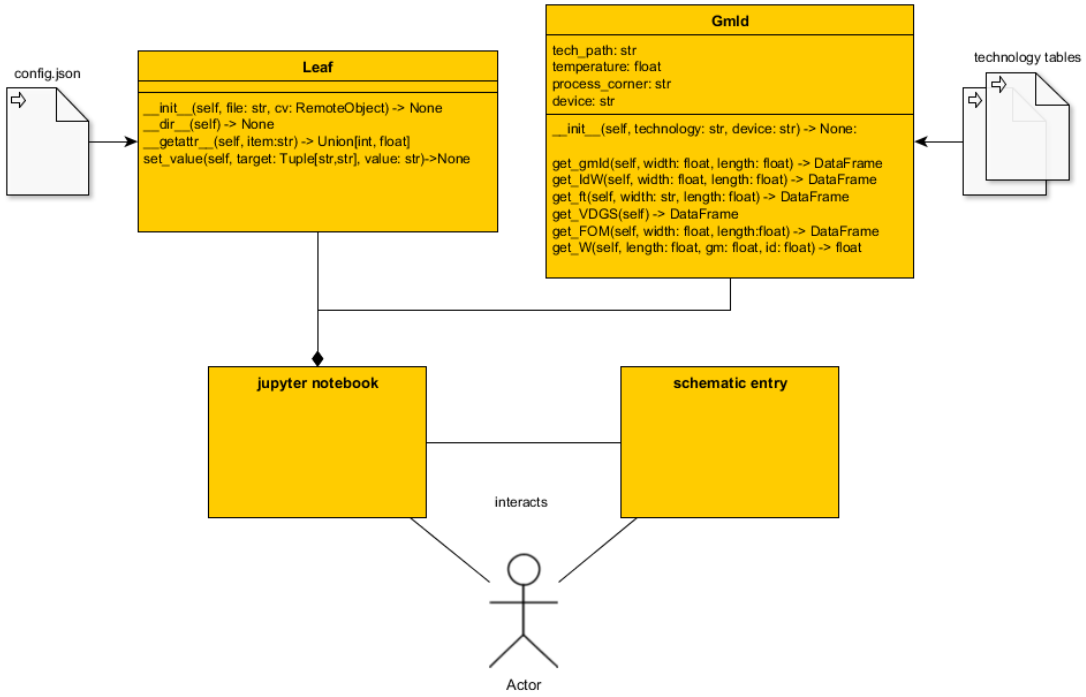


Figure 5.9: Interaction diagram of leaf sizing

prehensible for the circuit designer. To ensure design consistency the resulting internal design representation configuration files for each generated cellview are used. These configuration files include the evaluated parameters and local parameters of the corresponding system-level description and the evaluated ports.

To create and size a leaf cell two main tasks have to be executed. First, the schematic topology has to be chosen and drawn. Afterward, the circuit topology has to be sized.

In figure 5.9 the interactions for this workflow and some helper classes for the user are shown.

First, a circuit topology has to be created. For this two possible workflows for the circuit, designer exist. The circuit designer can use the schematic entry to draw an initial topology. This topology can later be modified in the circuit sizing step.
For often-used circuit blocks, a template schematic can be generated. The template schematic is created in the schematic entry with default values and connections names. In the next step, the template schematic can be translated to a SKILL script. This

SKILL script can then be injected into other cell views via methods inside of the python dimensioning script. These methods are included in the *Leaf* class.

When the topology is created or loaded a python sizing script must be written by the user. This can be done within a jupyter notebook which can be opened besides the schematic entry to interactively see the schematic manipulations. Jupyter notebook is a web application that allows live coding and visualization. In the background of the application runs a python interpreter and in a so-called notebook a collection of executable code blocks, markdown, and outputs. This allows developing code in an explorative way making it the best fit for circuit design. Furthermore, the markdown fields allow to include latex equations making it possible to write jupyter notebooks which execute code and document the reasoning for the sizing.

To simply access the internal design representation the Leaf class is used. With this ports and parameters can be simply accessed. On *Leaf* object generation the JSON configuration is read in and the parameters and ports are added as properties to the class. These can be accessed as shown in the example below.
The dunder methods *__dir__* and *__getattr__* are implemented to allow tab completion and access the in the config written values as object attributes.

```
# Create Leaf Object from config file
dco_core = Leaf(config_file)
# property access from the created config
dco_core.VCO_TANK_C_FIXED
dco_core.VCO_TANK_L
```

Furthermore, the Leaf class offers the set_value method to change parameters of devices in the targeted schematic. set_value can be used as follows

```
# set the parmater nf of device N0 to 20
dco_core.set_value(('N0', 'nf'), 20)
```

In the next two pages, a minimal sizing example for the DCO core can be seen. Besides the code cells, markdown cells are used to document the process. The example is reduced on some parts to fit on two pages.

In [ ]:
```
from skillbridge import Workspace
import GmId, Leaf, math
```

In [ ]:
```
dco_core_config = '/path/to/cellview/config.json'
```

In [ ]:
```
ws = Workspace.open()
cv = ws.ge.get_edit_cell_view()
dco = Leaf(dco_core_config, cv)
```

Maximum frequency:

$f_0 = \frac{1}{2*\pi*\sqrt{LC}}$

In [ ]:
```
dco.set_value(('L0','l'), dco.VCO_TANK_L)
```

In [ ]:
```
w = 10e-6
dco_parasitic_c = 200e-15
l = (dco.VCO_TANK_C_FIXED-dco_parasitic_c)/(w*unit_c)
```

In [ ]:
```
dco.set_value(('C0', 'w'), round(w,9))
dco.set_value(('C0','l'), round(l,9))
```

In [ ]:
```
max_freq = 1/(2*math.pi*math.sqrt(dco.VCO_TANK_C_FIXED*dco.VCO_TANK_L))
```

Oscillation Condition

Amplification

Tank negative conductance:

$g_{neg} = -\frac{g_{mn}+g_{mp}}{2}$

To ensure startup a multiplier for g_{neg} is introduced called α

In this example we will dimension $g_{mn} = g_{mp}$

$g_{mn} > \alpha * g_{tank}, \alpha \approx 2..3,$

Amplitude and current or voltage limited

Voltage limited:

$V_{ampl} < V_{dd}$

Current limited:

$V_{ampl} < \frac{I_{Bias}}{g_{tank}}$

In the current limited region the phase noise is worse than in voltage limited region

In [ ]:
```
def dco_core_get_tran_spec(Vsw, gtank):
    ibias = gtank*Vsw
    gm = 2*gtank
    return [gm, ibias]
```

Voltage swing, tank conductance from tank simulation and minimal length transistors:

$V_{sw} = 200mV$, $g_{tank} = 2mS$, $l = 20nm$

In [ ]:
```
gmid=GmId.GmId("gf22fdx")
gmid.device = "slvtnfet"
gm, ibias = dco_core_get_tran_spec(200e-3, 2e-3)
l = 20e-9
w = gmid.get_W(l, gm, ibias/2.)
nf = int(round(w/490e-9))

for device in ['N0', 'N1']:
    dco.set_value((device, 'l'), l)
    dco.set_value((device, 'nf'), nf)

gmid.device = "slvtpfet"
w = gmid.get_W(l, gm, ibias/2.)
nf = int(round(w/490e-9))

for device in ['P0', 'P1']:
    dco.set_value((device, 'l'), l)
    dco.set_value((device, 'nf'), nf)
```

Tail current source

Size voltage reference $nf_{ref} = 5$, $i_{ref} = 100u$

In [ ]:
```
nf_ref = 5
l_ref = 120e-9
iref= 100e-6
mult = int(ibias/iref)

gmid.set_device("slvtnfet")

dco.set_value(('N2', 'nf'), nf_ref*mult)
dco.set_value(('N2', 'l'), l_ref)
```

As we have demonstrated a modern toolchain was established to implement expert knowledge in an executable format. Furthermore, the sizing scripts can be implemented in a technology-agnostic approach and organized in classes and packages for reuse.

6 Layout Generation

The main challenge for layouts in advanced nodes is producing a manufacturable, DRC conform and robust design. Especially in modern nodes, the layout is dominated by silicon complexity as mentioned in chapter one.

Some additional challenges arise when considering the domain of High-Speed IO. To achieve high bandwidth IO interfaces high line rates are needed and high pin densities. The necessary pin densities lead to area IO. This means compact designs are needed to keep the parasitics low for the High-Speed circuitry. Furthermore, the silicon IP has to fit between smaller pin pitches. It is not desirable to become limited by the SERDES IP area.

In the first part of this chapter general guidelines and best practices for layout in advanced nodes are discussed. These guidelines should be encoded in this chapter implemented layout generators whenever possible. In combining the best practices with the expert knowledge the result will be a more uniform and integrative layout block.

Full layout generation is not a realistic goal that can be achieved directly. In this work, a step-wise approach was chosen. The first step is the generation of elementary cells which can be extended by additional generators to build full layouts which are with additional generators extendible to full layout generation.
The core concept of this chapter and the first step towards full layout generation is the partial layout generation of layout sub-blocks. Reoccurring primitive cells in the full custom domain were identified and implemented as parametrizable layout blocks.

Definition 10. *Elementary cells* are small reoccurring design blocks in the layout consisting of only a few devices. Examples for *elementary cells* can be transistor arrays, transistor pairs, or passive/active loads.

Elementary cells for the layout do not necessarily describe analog functions but rather a common interconnection approach within the context of a few devices either passives or transistors.

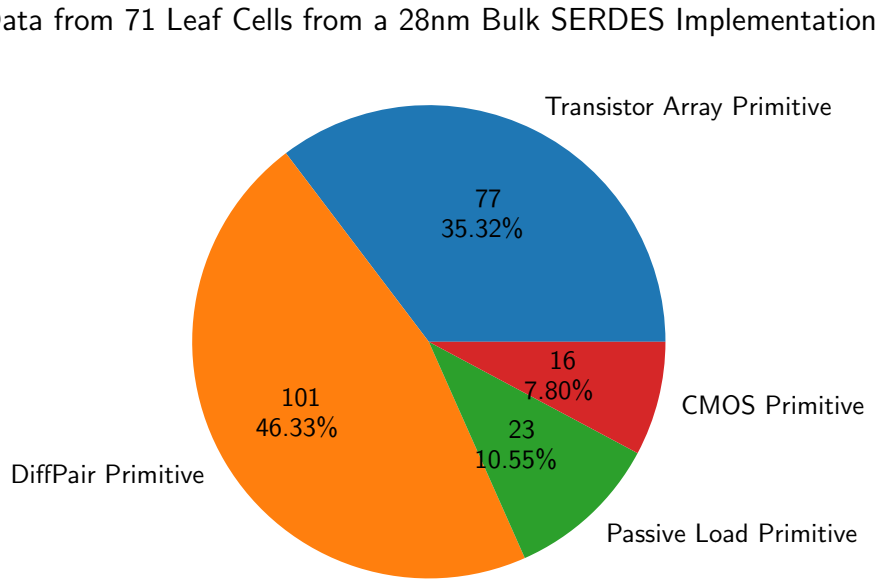


Figure 6.1: Determined reoccurring layout blocks

During the schematic phase, these elementary cells can already be defined. In the schematic phase, it may be that some parameters of the elementary cells are not known yet. In the layout phase, these parameters are added to generate the desired layout part. The elementary cells are implemented in a versatile way to be used as building blocks in most full custom circuitry and build up the analog building blocks.

Definition 11. *Analogue building blocks* are circuits build-up from elementary cells. For example, a CML Buffer consisting of the elementary cells transistor pair, transistor array, and passive load.

Within a chip design project of a 28nm bulk SERDES it was noticed that the layout took too much design time and that some layout blocks are done repeatedly.

It could be determined that transistor pairs, transistor arrays, passives, and a CMOS style layout were the most reoccurring typical small layout blocks. In figure 6.1 71 leaf cells of the design project were analyzed and the potential elementary cells were noted. The most common repeating layout implementation is the transistor pair (101) followed

by the transistor array (77). CMOS-like cells were identified 16 times and passive loads 23 times. In this analysis layout, leaf cells for fill and decap were not examined. For the transistor array, elementary cells only obvious transistor arrays were count and often interpreted as a single elementary cell i.e. for the main current mirror. This results in a slightly more pessimistic estimation.
Not only the repeated layout of these blocks are problematic. Another disadvantage is that these multiple layouts of the same element type differ from each other. On the one hand, because a common approach only was developed in later design blocks over time. On the other hand, since multiple layout designers implemented these blocks in slightly different ways.

The goal in this chapter is to implement these elementary cells to reduce the 217 different layout blocks to 4 parametrizable blocks and thus increase design efficiency, reduce design turn around cycles and increase uniformity in the layout.

This approach will be interactive as the schematic/ layout designer will choose which cells should be made up from the elementary cells. In figure 6.2 an example of a schematic build-up from elementary cells can be seen. The chosen example is a delay cell of an injection lock ring oscillator and the potential elementary cells are marked. All current source transistors are marked for a single transistor array elementary cell. If the layout designer wants to place each current source independently a transistor array constraint for each can be defined. For the two differential pairs, the input stage, and the injection stage a transistor pair elementary cell can be used. For the cross-coupled load and the PMOS load again the transistor pair elementary cell can be used since they can be interconnected with the same basic idea but then only differ from their input connections.

The generation of elementary cells was initially based on SKILL PCell which were developed with the PCell Designer. In this approach, each PCell existed as a symbol, schematic, and layout PCell and could be placed in the schematic as elementary blocks instead of using the native PDK transistors. This approach will be discussed in the first section of this chapter.

With the initial PCell implementation of these elementary parametrizable layout generation cells, several issues arose. Based on the lessons learned a technology-agnostic layout generation framework is implemented. This approach does not offer elementary schematic blocks as the PCell approach but rather lets the circuit designer add con-

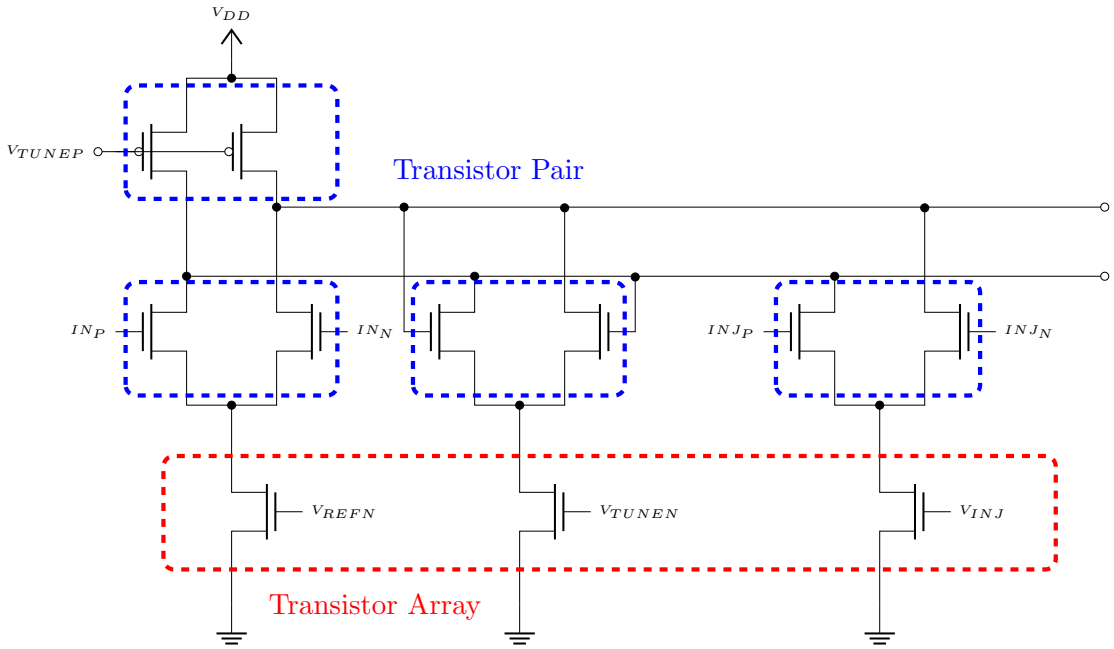


Figure 6.2: Elementary cells example ILRO delay cell

straints to the schematic which mark the cell for generation in the layout step. With this framework, elementary cells and other generators are built. In the second part of this chapter, the implementation of these will be discussed.

6.1 Layout Approach

The increasing DRC and DFM rules and working with multiple people on a design make common layout guidelines and best practices necessary.

It is important to establish a layout approach for everyone designer to follow. The goal is on the one hand to simplify the layout process and on the other hand allow simple integration of different layout blocks in its parent hierarchy.
A solution for this is a track-based layout approach. In a track-based approach for each metal layer, a track is defined. The layout designer work on the defined tracks. Tracks are defined per metal layer and set a direction, track width, distance to neighbor tracks, and optional track offset on which the tracks start. With this approach, the layout becomes simpler and more uniform.

Two arising issues in modern nodes will be discussed in detail which are lithography,

regular design, and electromigration. Both are discussed in context to our technologies. For these rising issues, some analysis approaches for early evaluation are introduced. Within this section, an estimate will be made on how to incorporate these beneficially in the design flow.

Afterward, the chosen best practices in terms of track definitions and default device parameters are discussed and some alternative approaches are reviewed.

6.1.1 Rising Layout Challenges

To define best practices some aspects of today's challenges for cell level layouts have to be introduced. From these analysis methods and best practices for general layout design and layout generation can be derived.

Traditionally, for cell level layout it has to be checked that the layout meets its area constraints, does not block too many layers, and still meets its specification after parasitic extraction. Furthermore, in advanced nodes, several other aspects like density, regularity, and electromigration robustness have raised in importance. For these layout aspects simple analysis often does not exist.

In the following sections, some analysis and best practices are derived. Furthermore, for some aspects, it is discussed if the best practices can have a measured impact and how they might be enforced.

Lithography Friendly Layout and Regularity

An increasingly complex and important topic is lithography. In this subsection, the basic concept of the lithographic-friendly layout is introduced. An assessment of the impact on circuits is given based on literature and our used technology PDK and the included simulation views.

With shrinking nodes lithography has become more and more challenging since the wavelength scaling of the light source has stagnated at $193nm$. The reason for this is the absorption edge of air which becomes significant below $185nm$. Resulting from this today's feature sizes are a fraction of the light source wavelength. Today several other methods are used to further decrease the resolution like multi-patterning or special lenses

used in purified water.

The main problem for ASIC layout design is the impact radius. The impact radius is the radius around a shape where this shape causes interaction between other shapes due to the Lithography. In a modern node for a $193nm$ light source which corresponds to an ArF Laser which is commonly used for nodes below $50nm$ the interaction radius is around $500nm$, from the Introduction of [58]. This effect results in widened Worst and Best-Case corners. This further results in more complex DRC or DFM rules for the technology node. Lithography is one of the major contributors to chip variations.

One approach to reducing the effects of lithography is the use of lithography-friendly patterns and with that a regular layout. Furthermore, a regular layout meeting density rules should also be better in regards to the variation from chemical mechanical polishing (CMP).

Strictly regular layouts have become more and more popular. An especially prominent example is the Standard Cell design in advanced nodes either to improve device characteristics i.e. leakage [59] and to reduce variability. Here the poly layer is highly regular which means one orientation, one gate length, and one poly pitch within a device library.
Furthermore several layout approaches for yield optimization based on regular layouts are introduced in [60], [61] and [62].

In [60] a regular approach for Standard Cell designs is evaluated. It uses a reduced Standard Cell Library to increase regularity on the macroscopic level and the other hand a set of lithographic-friendly layout bricks.
In an ARM9 test design from [60] it is shown that the variation of the poly gate across chips could be reduced by a factor of 2. This approach makes use of "pushed" design rules enabled through the brick-based design which additionally results in a 15% smaller design. Unfortunately to use pushed rules strong cooperation between foundry and user must exist.

[61] discusses the importance of regularity and defines an impact radius in which shapes are influenced by other shapes within the radius. This impact radius becomes bigger in smaller technologies since the gap between light source and structure is increased. Further, the paper discusses so-called lithographic dream patterns and how to optimize the

lithographic process to generate them with high accuracy. Afterward, layout methods are discussed which could be used in such a lithographic process were implemented.

In [62] a via configurable transistor array (VCTA) is introduced which is highly regular and from it, digital cells can be derived. Furthermore, the work introduces a regularity metric and a physical synthesis tool working with the VCTA cells.
It was shown that this approach increased the initial yield of a design because of regularity when comparing to a Semi-Custom or Full-Custom approach. Disadvantages are the impacts on the area, energy, and delay. It is noted that these drawbacks might be solved in cooperating with the foundry to allow pushed rules.

After this summary of the academic state of the art practical approaches have to be derived. A lot of the proposed approaches are not usable when the lithographic process cannot be modified and close cooperation with the foundry is not available. Also defining stricter rules to create a more regular layout is not constructive if these cannot be associated with a predictable impact.

The most ideal approach would be analyzing the layout and from this creating new tightened corner views. For this tools exist which can create contour files of a layout for example the Cadence Litho Physical Analyzer [63]. The contour data can be combined with the extracted layout results to create new lithographic-aware corner views.
Unfortunately, both technologies we use do not provide the needed lithographic technology files for this workflow.

Nevertheless, there are several other ways to at least get an estimate about the yield a layout will accomplish. For smaller technologies often a DRC plus deck exist which mainly checks for lithography-friendly layout and gives the layout a score. A specific score must be reached to manufacture the chip. This score is loosely associate with the potential yield of the design. For this, the DRC plus deck must be run, which includes pattern matching for worse lithographic patterns.

As a best practice, it can be derived that a regular layout is desirable. Unfortunately, these best practices can not be matched to a direct impact on the design but rather to a better DRC plus score in the end.

EM Limit

Another important part of the layout validation is electromigration (EM). To validate for electromigration a back annotated simulation is run to create the needed EM data. Afterwards, the in the PDK included EM deck is used to check if the current limits of the shapes are exceeded. These tests are done late in the design since a complete layout has to exist. This section introduces an approach to get early estimates if shapes are EM robust. The results of this can be used together with expert knowledge to get a good estimate.

In chapter four the theoretical background for today's non-linear EM-rules is given. These determine the maximum allowed average current of a shape.

The maximum allowed average current of the shape can be changed by modifying the geometry of the shape. As described in chapter four due to several effects (immortal wire, ...) widen the shapes in every case can result in worse EM performance. This especially is the case for shorter length shapes in our technologies this corresponds to a length below $8\mu m$ and thus is relevant for most shapes especially in block-level layout and the lower metal layers which tend to be shorter.
With the change of geometry parasitics of the shape change accordingly. While the capacitance change can not reliable determined just from one the independent shape geometry, without knowledge about neighboring shapes. In contrast, the resistance of the shape is solely depending on the geometry and independent of neighboring shapes. The change to higher resistance of a shape can be seen as an unwanted effect since it will impact most circuits negatively.
In this initial approach to determine the EM robustness of a cell level layout the capacitance as a factor will be ignored.

Every given geometry shape should be optimized in the following way. $\frac{\Delta I_{avg_{max}}}{\Delta R}$ should be maximized. The original area of the geometry should not be succeeded. The resulting optimized $\frac{\Delta I_{avg_{max}}}{\Delta R}$ can on the one hand be used to debug critical shapes and to fix them. On the other hand, it can be used as a layout metric for EM optimal shapes. In this approach the information about the current is missing thus it does not need simulation and will give a fast initial overview over a layout block.

The function to calculate the maximal allowed average current can be derived from

$n \leftarrow 1$

while $W_{min} \leq w$ **do**

$W_{remove} \leftarrow \frac{D_{min}*(n-1)}{n}$

$\Delta R_{max} \leftarrow \Delta R_{max} - \frac{W_{remove}*n}{W_{original}}$

$w \leftarrow \frac{\lfloor(\frac{w}{n} - W_{remove})*100.0\rfloor}{100.0}$

$X_0 \leftarrow \frac{\lceil w*(1-\Delta R_{max})*100.0\rceil}{100.0}$

$x \leftarrow \{X_0, X_0 + 0.01, X_0 + 0.02, ..., w\}$

$y \leftarrow \frac{I_{avg_{max}}(x,l,layer)*n}{I_{avg_{max}}(W_{original},length,layer)} * \frac{x*n}{W_{original}}$

$Y_{best\_new} \leftarrow max(y)$

if $Y_{best\_new} > Y_{best}$ **then** $Y_{best} \leftarrow Y_{best\_new}$ **end if**

$n \leftarrow n + 1$

end while

Figure 6.3: Algorithm to determine the EM robustness of a shape

the EMIR rule deck. It is depending on T_{life} which determines the meantime of failure. The resistance change is directly proportional to the width change of the geometry. In this approach, the length of a shape is not changed.

The algorithm 6.3 for this first tries to narrow the shape until the ΔR_{max} is reached for each step $\frac{\Delta I_{avg_{max}}}{\Delta R}$ is calculated and the maximum is saved. Afterward, the shape gets split up into two shapes and is resized again. This is done until the defined minimal width W_{min} is reached by the split shapes.

To understand this in detail we assume the following example. A shape with a length of 2 um length and 0.62 um width is given.
The function $I_{avg_{max}}(w)$ for a constant length of 2 um is shown in 6.4. The colored regions are the in the rule file divided rules which result in the discontinuous function shown.
Our original shape starts with $I_{avg_{max}(0.62\mu m)} = 0.56$. For this example we will alow a $\Delta R_{max} = 0.2$ and a resolution of $10nm$. In our algorithm first W_{remove} and with it ΔR_{max} is calculated. At the start for $n = 1$ these metrics are not changed. Now the search space to find the maximum $\frac{\Delta I_{avg_{max}}}{\Delta R}$ is calculated in this example with a resolution of $0.01nm$ 0.5...0.62.
In this iteration, a with of 0.52 is found to be ideal since it allows 2.09 times the current

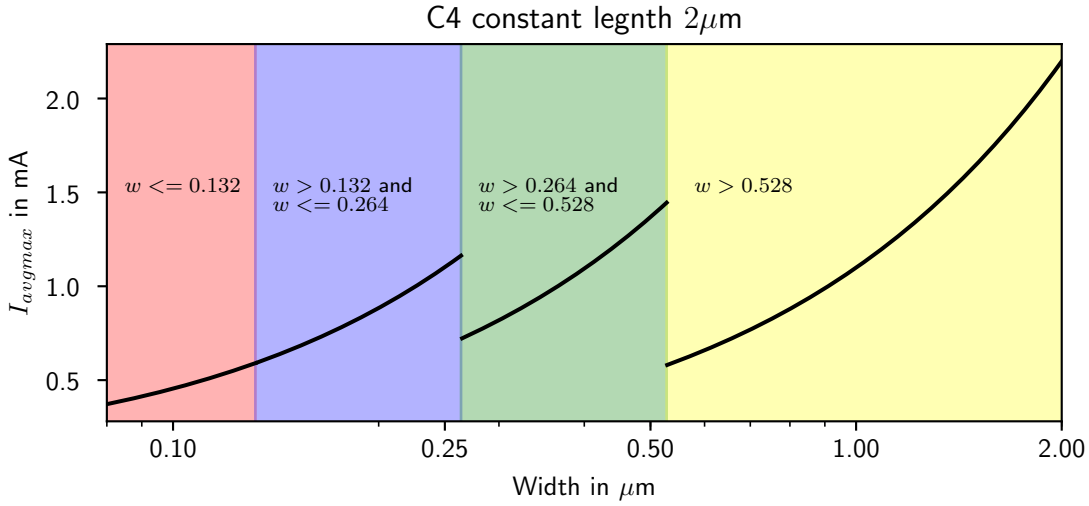


Figure 6.4: Example Tech Dependent $I_{avg_{max}}(w)$ for 2 um

with a resistance change of 16.13%. Each iteration can be seen in figure 6.5. With $n = 3$ the break condition is reached since the resistance change is above the constrained 20%. An ideal geometry is found in b at $n = 2$ which allows 3.36 times the current with also a resistance change of 16.13%.

Density

Density is an increasingly important metric the multiple interacting density rules in a PDK are difficult to match. One problem is that the density rules are defined for different layout windows. If the layout block is below this window the corresponding layout

(a) $n = 1$
$width = 0.52$
$\frac{I_{avg_{max}}}{I_{avg_{max}orig}} = 2.09$
$\Delta R = 16.13\%$

(b) $n = 2$
$width = 0.26$
$\frac{I_{avg_{max}}}{I_{avg_{max}orig}} = 3.36$
$\Delta R = 16.13\%$

(c) $n = 3$
$width = 0.09$
$\frac{I_{avg_{max}}}{I_{avg_{max}orig}} = 1.81$
$\Delta R = 56.45\%$!!!

Figure 6.5: Example $2\mu m$ length, $0.62\mu m$ width EM optimization

will not be checked for the rule. To find and check critical density parts in the layout early custom scripting checking these is introduced. These check the most critical density rules even when the window size is not reached yet and it is possible to check the accumulated density rules over multiple layers.

Furthermore, the density gradient as a metric is interesting since it should ideally be constant. The DRC rules in modern nodes are only able to check density gradients between specific density windows so a gradient plot can give a good overview early of possible problems.

For generator and best-practice approaches, the density has to be taken into account. Especially for the track setup, a dense power grid should also be density clean, and layouts from potential generators should be density friendly.

6.1.2 Best Practises for a DFM Driven Layout

The main idea with DFM Driven Layout is to define a layout environment and best practices which will result in a DRC clean and EM robust design. Furthermore, this layout environment and rules should result in simple integration of the cell in the design hierarchy. In the following, some approaches which will be used in layout generators are presented and discussed.

One aspect is track-based design. Here a grid of tracks is defined for everyone taking part in layout design. This has several advantages as the resulting layout blocks will be simply integrable in the system level and result in a more overall regular layout.

This can be further increased when introducing specific tracks for specific aspects of the design for example a track definition for power, bias route, and HF route, and more to give layout designers a common approach for the associated task.

The chosen tracks and design grid for this approach are based on the one hand to achieve sufficient density and on the other hand to result in aligned Vias resulting in via towers from the upper coarser metal grids to the lower metals. Furthermore, it is necessary to check that the different track definitions as power or bias route have a common multiplier and will not result in problems when used together.

In the below table an example track setup is shown. In this example, only the most

commonly used tracks are shown with the cell track and the power track with VDD and VSS tracks. It should be noted that each layer has a corresponding orientation which is also defined in the track but not shown in the table. In the example, it can be seen that the power track is separated into two tracks with different offsets and a stride of two. This results in a layout-wide defined power grid for every layout block and every designer increasing simplicity on the integration of layout blocks.
One of the main aspects when defining tracks is the resulting metal density when the tracks are filled. This density must meet the PDKs minimal and maximum density rules. The densities for VDD and VSS track per layer have to be added since they alternate.
Another challenge for the track setup is that it should result in via tower laying above of each other especially for the power tracks. This, on the one hand, increases EM robustness and on the other hand, simplifies the connection from the power grid to layout elements. For the cell tracks, it has to be taken into account that it is used to fan out multiple signals at once and thus should result in compact bundles of wires. Additionally, the vias on the cell track have to be able to be placed in an alternating manner and they should always fit at least 2 vias to match DFM constraints.

Besides the track definitions, some defaults should be set for the PDK devices used. In the Semi-Custom design for this Std-Cells in rows with standard heights are used. While this row-based design is becoming increasingly popular in the full-custom flow for FinFet based designs in our case we define PDK defaults to further increase a uniform design but keep some degrees of freedom to achieve compact and robust designs. Thus for the

Layer	Type	Width[μm]	Distance[μm]	Offset	Stride	Density[%]
M1-M2	Cell	0.040	0.125	0	1	32
M1-C1	VDD	0.130	0.25	1	2	26
M1-C1	VSS	0.130	0.25	0	2	26
C1-C6	Cell	0.06	0.125	0	1	48
C2-C3	VDD	0.3	0.5	1	2	30
C2-C3	VSS	0.3	0.5	0	2	30
C4-C6	VDD	0.62	1	1	2	31
C4-C6	VSS	0.62	1	0	2	31
IA-IB	Cell	0.36	1	0	1	36
IA-IB	VDD	0.62	1	1	2	31
IA-IB	VSS	0.62	1	0	2	31

Table 6.1: Example track setup for cell and power tracks

transistors, standard finger widths for the layout designer to use are defined. Additionally, default parameters for analog and digital transistors are set up. These parameters are set to fulfill recommended DFM options and will result in the gate contact laying on the M1 cell tracks.

These tracks and default transistors should be used whenever possible but there are some exceptions for example extremely dense layout parts, high current shapes, or special sensitive analog blocks. If such a block is designed at least the resulting pin shapes or at some point the shape structures have to end on the defined global grid, so the block can be integrated at the system level. These defined best practices have also be encoded in possible elementary cell layout generators.

6.2 PCell Designer based Implementation of Primitive Cells

For a first implementation for the elementary cells, the Cadence PCell Designer was chosen. The PCell Designer allows to program PCells with higher-level functions in a graphical way. The Pcell Designer is described in detail in the State of the Art chapter.

The first two implemented elementary cells are the transistor pair and the transistor array. Each is explained here with its challenges and approaches for different design problems.
The PCell Designer can be seen as a programming tool that generates skill code from its graphical description. This graphic representation promises to offer the same concepts as modern programming languages as in example methods, recursion and inheritance .
One design decision is to compose the PCell elementary cells hierarchically.
Initially in this section, the vendor transistors will be introduced and discussed.
The base for all these cells should be the BASIC-FET. Its function is to encapsulate the original vendor transistor PCells and their behavior. This cell is then used to build all other PCells from it.
Furthermore, every cell will inherit from a base cell. In this cell, the version of all PCells is set. Afterward beginning with the BASIC-FET each PCell implementation and setup will be described in the following subsections.

6.2.1 Vendor Transistor Pcell

The vendor PDK already comes with a transistor PCell which is used to build up more complex configurations. Not using the vendor PCell would result in increased complexity

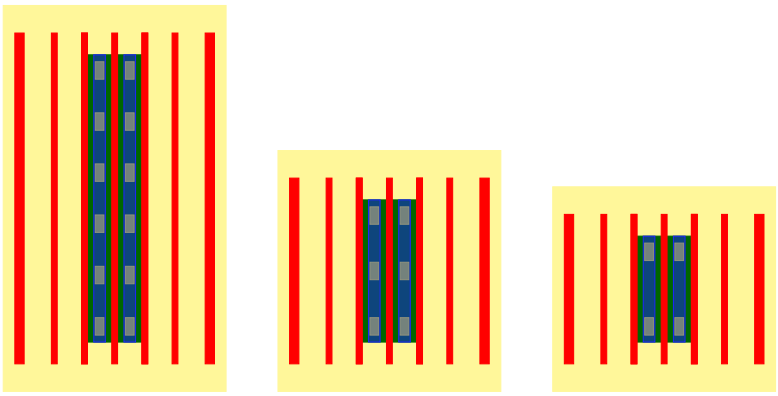


Figure 6.6: PDK transistor with default parameters and the chosen preferred finger widths

to build transistor-based PCells. Not only that the vendor PCell has a lot of configurations that would have to be implemented, furthermore these parameters depend on each other and trigger callback functions to update the dependent parameters. To further complicate the matter these parameters are not always layout parameter which are updated but also parameters used in the spice model or to draw layers for the DRC, LVS or parasitic extraction.

This is the main reason to build upon this cell. Furthermore, this will also be a future save variant since the PDK updates will automatically be applied to dependent cells. Also, it uses already existing cells and limits the implementation effort.

The vendor PCell is very versatile since it has to be usable for every possible circuit. The PDK transistor PCell has around fifty CDF parameters. Some only impacting the simulation model and some impacting the layout. For example to create an abutable transistor first a flag has to be set to allow to modify the dummy configuration. Next poly dummies have to be disabled. For this, the parameters (Dummy config, Number of Dummies, and Stop Dummies) have to be set. Instantiating this PCell with the correct parameters is the goal for the BASIC-FET PCell.

6.2.2 BASIC FET

The BASIC FET is a helper cell, which is not intended for direct use. It provides the different basic transistor structures needed by the elementary cells.

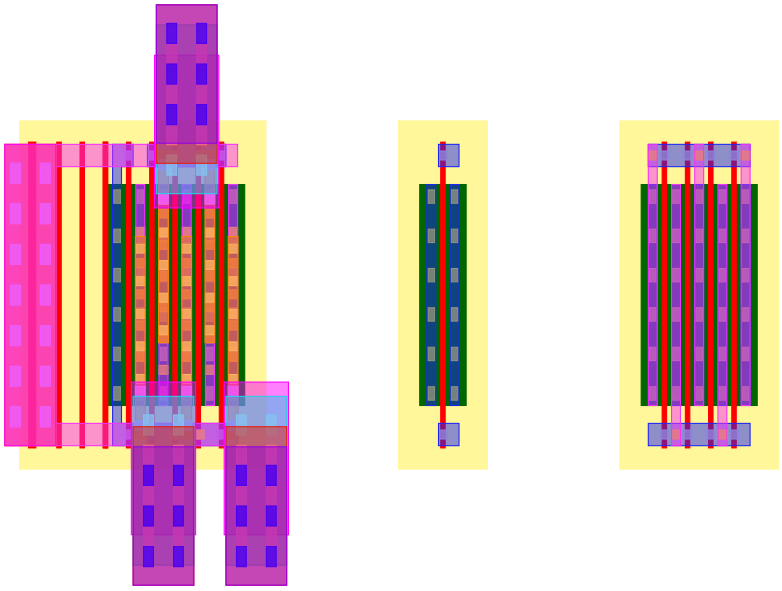


Figure 6.7: BASIC-FET in the configurations active, left dummy and middle dummies

In figure 6.7 the most basic configurations are shown.
On the left, the active FET with a dummy transistor and a gate connection on the left side is shown. This configuration is used as left and as mirrored version as the right active part of the transistor pair PCell. It also exists in a variation where the gate connection is not drawn or drawn on both sides, which is used as a building block for the transistor array elementary cell. The same applies to the dummy which also can be drawn on both or only one side.
On the right, a middle configuration is shown with all transistor terminals are connected to the dummy net. This configuration is used in transistor pair as middle dummy between the two active devices. The actual dummy terminal and connection are drawn in the transistor pair PCell since it is dependent on some of its parameters.

$$\begin{pmatrix} LVT \\ SLVT \end{pmatrix} \times \begin{pmatrix} Core\ Device \\ 1.2V\ EG \\ 1.5V\ EG \\ 1.8V\ EG \end{pmatrix} \times \begin{pmatrix} NMOS \\ PMOS \end{pmatrix} \tag{6.1}$$

The three possible gate width configurations are chosen to create a more uniform design and with this configuration, the vendor PCell created gate stripe already lies on our defined design tracks. With the in the above section defined defaults to create more

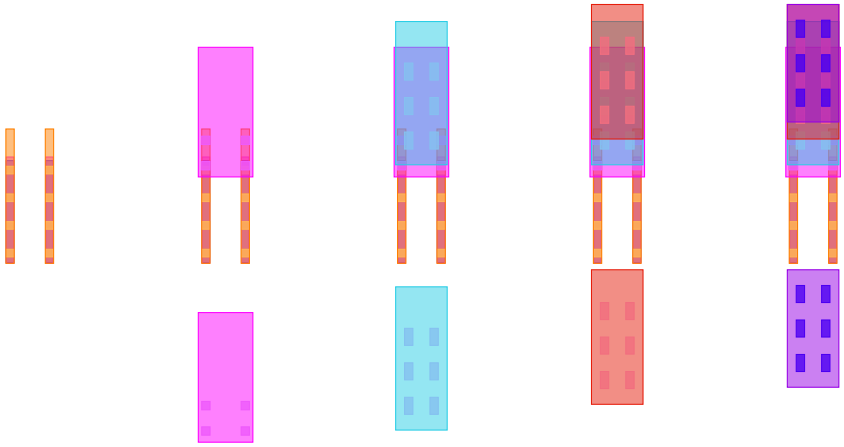


Figure 6.8: layer structure for drain and source clips

uniform layout blocks.

In figure 6.8 the layer structure of the drain and source clip for active devices are shown. These structures allow high maximal current density. The alternative configuration is connecting drain or source with a stripe connecting down to the corresponding M1 metal. The advantage of the clip structures is that each element has a small equivalent length, allowing a high average current for commonly used gate lengths, while for a strip the length would depend on the number of the fingers of the transistor. And thus with a greater length result in a lower I_{avgmax}. The upper layer of the clip structures can be configured. When the clips are configured to connect to C5 they will lay directly on the IA grid and can be connected to IA. I metal has 14 times the thickness of C metal and allows reliably very high currents EM wise.

6.2.3 Transistor Array and Transistor Pair PCell

Transistor Array and Transistor Pair PCell implement the elementary cells by composing them from the BASIC-FET PCell with passing the corresponding parameters. For each PCell, a schematic, symbol, and layout view is created with the PCell Designer. The PCell designer offers the functionality to create a graphical equivalent to methods that then can be used as a custom command. Inheritance does only work for cells with the same views meaning schematics can only inherit from schematics, symbol from symbol, and layout from layout. This makes it difficult when multiple cells over different views

but share the same traits as these must be implemented manually for each cell and kept consistent by hand.

In figure 6.9 on the right, some transistor array examples can be seen. The transistor array PCell allows selecting one of three defined finger widths for the included transistor. Furthermore, the number of fingers and the length of the transistor can be set. Additionally, a transistor type and a VT variant can be set. Each of these parameters is passed through to the BASIC-FET. For the transistor array rows and columns can be set. Row and column act as a multiplier for the other parameter set transistor resulting in the defined array. In the example in a single device transistor array is shown with guardrings. In b) a 2x2 transistor array and in c) a 2x1 transistor array without guardrings.

The Transistor Pair uses all the above introduced different modes of the BASIC-FET with constructing a composition of left dummy, left device, middle dummies, right device, and right dummies. On the right several examples of this can be seen. As for the TransistorArary, the transistor parameter can be set. In this implementation, the second transistor in the pair is always mirrored and the same as the first one. Always resulting

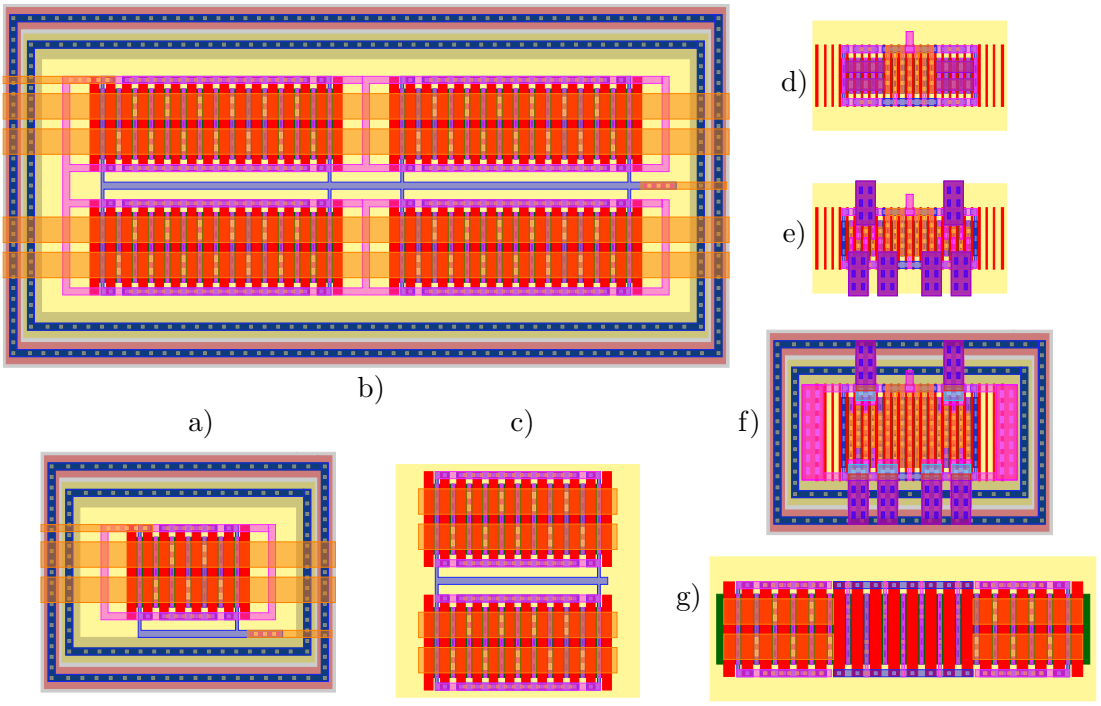


Figure 6.9: PCell elementary cells left: Transistor Array, right: Transistor Pair

in a symmetrical pair. In d) a variant with stripes on C5 is shown. In e) a variant with clips. In f) with clips and guardrings. In g) a transistor pair for IO transistors with stripes is shown.

6.3 A constraint based layout generation approach

From these lessons learned from the PCell elementary cells, a new approach was derived. This approach should succeed the PCell approach by keeping its advantages and resolving the disadvantages. The main challenges as the side effects and no technology independence will be discussed in this section. Afterward parametrized shape generation is a topic since the programmable build-up of geometries is an important topic for a layout generator. The better usability and increased code reuse in the generators are topics in the following implementation sections.

For each of these aspects, different proposals are discussed and will be the focus of a resulting constraint-based layout generation approach.

6.3.1 Side-Effects

One issue which arose when using the PCells in bigger designs were side effects with other skill code in the design. This issue did not occur during the design phase but rather on streamout resulting in destroyed .gds files. This error only occurred in some complex combinations between vendor PDK, PCell Designer, and custom PCells. After some experimentation, it could be determined that in a specific execution order of the PCells some would evaluate with errors. Before each streamout in layout or netlisting in the schematic an evaluation takes place. The evaluation sequence from which this issue depends is not documented and it seems that this cannot be changed. It is not possible to further debug the problem since a lot of third-party code is involved. Additionally, this should not be a problem since the PCell Designer Code should be completely encapsulated. In this error case, it is not which may hint at a bug in the PCell Designer. Since the example to reproduce the error involves multiple large layout blocks a minimal test case to create a support request can not be derived.

In newer versions of the PCell Designer, there is the concept of an AppCell which is a PCell without instance. It draws the layout flat in the entry and thus it will not be re-evaluated by the streamout or netlisting. This approach allows the user to change some parts of the AppCell when it does not fit completely. A flat layout generation seems

all in all an approach with advantages but a disadvantage which has to be overcome is that regeneration of cells and a clean up of cells is needed. It is not uncommon that elementary cells are improved and that the user wants to regenerate already existing layout blocks.

6.3.2 Technology Independence

The PCell Designer approach is not technology, stack-up, or node independent if any of that changes all cells have to be reimplemented. A technology-independent description of such cells is desired. Several approaches for technology-independent descriptions can be found in literature. One example is [64] where a virtual design grid is established and abstract definitions for different well contacts and P and N-type transistors are introduced. When comparing the aspect of the paper with today's technology some aspects come short i.e. the buried contacts in FDSOI. Furthermore, today's PDKs became more complex with modern nodes i.e. design rules and the existence of vendor PCells. There are several good approaches in this work that are still valid today, i.e. the virtual grid and the well abstraction, grouping, and generation. Furthermore, it should be noted that the goal of [64] is to describe STD-Cells in a tech-independent way rather than arbitrary circuits. This way some aspects can be used today but many come short for a general approach.

In the following section, it is evaluated what is needed to achieve a technology-independent description.

In a technology, there are a lot of aspects that are specific to the technology. Design rules, vendor-specific PCells with vendor-specific parameters, stack-up, and layer naming. Not every detailed information of the technology has to be abstracted only the parts necessary for layout generation are needed. From a general approach for layout generation, the technology-dependent information should be derived.

When working flat on the layout to avoid side effects some other advantages come with it. For one this approach is based on already existing instances on which constraints are applied. This means the generator does not have to create these instances but rather set an intention or constraint which has to be attached to the instance or instances. Working on constrained instances introduces automatically some technology independence since the instance can be swapped and as long layout generation works on the same parameter set the result should be a valid layout. This indifference to the Cadence PCell approach

means there are no conditional cases that have to be taken into account for different device types i.e. LVT, SLVT or high voltage transistor types since they largely share the same parameters and terminal names.

Layout generation independent from the instances has to work on a given layer list. This layer list is highly technology, process and stack-up dependent. Since the layer list is a commonly shared aspect in all cells, it is one of the most important abstractions. If the abstraction of the layer list is done right a lot of otherwise existing dependencies can be removed. The abstract layer list has to include in the generator needed layers. There are a lot of helper layers for DRC and LVS not all of them are needed in a generator. Nevertheless, it must be possible to define them if needed. The goal is to find a general technology and stack-up independent layer list which is reduced to only the necessary layers.

The general layer list should consist of general layer names being converter to the corresponding specific layer name of a target technology.

6.3.3 Programmable Shapes

An important task to accomplish is the programmatic description of layout structures. One of the core concepts in the PCell Designer for shape construction is the geo expressions, which lets the user define an expression that will result in an output polygon or list of polygons. The geo expressions are very powerful and a lot of tasks can be accomplished in multiple ways. Geo expressions in the PCell Designer are set up graphically due for this reason they become cumbersome to set up fast. An important point of the geo expressions is that most expressions are based on querying existing shapes in the layout. If the layout shape changes the result of the geo shape will change accordingly resulting in parametrizable constructs.

To find a fitting solution different approaches for geometric constructions were evaluated. From one of these evaluations, the simple-geometry package was created. *Simple geometry* is an initial implemented python package for axe-aligned geometry descriptions. Similar to the skillbridge the simple-geometry package was released as an open-source project on GitHub [65].

In advanced node technologies for shapes only often only vertical and horizontal shapes are allowed. From this, the decision was made the geometry descriptions should be

axis-aligned as this prerequisite further simplifies most methods. A second prerequisite for further simplify is to only allow rectangles as polygons can be build up from these rectangles.

The simple geometry package introduces classes and types for points, rectangles, path segments, groups, and a default canvas.
The Point class offers a way to define a 2d point in space and is build up from the x and y properties which represent the coordinates. The Point class has an alias called Vector to allow to document in the code the intend of the operation. For the Point/Vector class multiple operations are implemented as element-wise division, multiplication but also a dot product, and many more.
The rectangle class is used to define rectangles and introduced properties as x and y coordinates but also a width and height. Furthermore, the corner handles can be read with top_left, top_right, bottom_left, and bottom_right. Corner handles return a Point. Besides corners and edges exist which are left, right, top and bottom. Both edges and corners can be manipulated resulting in a modified Rect object. The Rect class further implements useful methods as intersection returning the intersection as Rect Object or None if there is no intersection, is_inside_of returning a bool or copy, translate and union.
The Segment class implements a rectangle that has a direction and can be defined with a starting point, endpoint, and width. Segment objects and Rect objects can be translated into each other. Furthermore, the Segment class adds the Direction property.
The Group class allows to group the geometry objects. Furthermore, the group can be modified as in example it can be mirrored, copied, or translated.

```
from geometry import Rect, Canvas

big = Rect[100, 100, 'blue']
small = Rect[50, 50, 'red']
small.translate(center=big.top_right)
tiny = big.intersection(small).copy('green')

c = Canvas(width=200, height=200)
c.append(big)
c.append(small)
c.append(tiny)
```

The default Canvas class takes the geometry description and outputs an HTML visual-

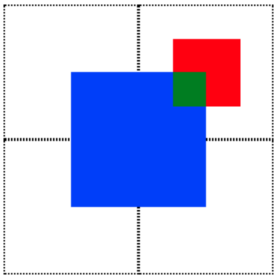


Figure 6.10: Output simple-geometry example [65]

isation.

The above-shown example is modified from [65]. Here three rectangles are defined. The big and the small rectangles are defined directly with a Rect object. Afterward, the small rectangle center is translated to the big rectangle top_left. The tiny rectangle is constructed from the intersection of the big and the small one.

With rodlayout [66] a canvas backend for the Virtuoso layout entry was implemented. rodlayout uses the skillbridge to communicate with Virtuoso. Instead of giving geometry objects colors, a layer purpose pair can be set.

The simple-geometry package allowed to intuitively describe geometry constructs. But when used for layout generation some disadvantages were discovered. While the construction of the objects worked seamlessly the interaction with already existing shapes, groups, and instances in the layout was lacking. For the user, it was often unclear how to manipulate which object, as there was no common method, and the simple-geometry description behaved like a virtual copy.

Due to this in the XCell approach which will be detailed in the next section, the separation between geometry construct and canvas was removed. Instead, a defined interface to query and draw layout objects was introduced with the Viewport. The advantages form simple-geometry are implemented in the Box class to give a clean interface to construct parametrizable interdependent constructs from queried boxes of layout objects to the Viewports draw methods.

6.4 Xcells

The implementation of this constraint-based layout generator framework is XCell. This Framework is developed with the lessons learned in mind. It offers all needed functions to implement layout generators and abstracts the design environment in use and the specific technology.

In this section, at first, the core concepts and structure of the layout generator framework are introduced in a top-down manner.

The technology abstraction is one of the most important features and will be discussed in detail in a separated section.

Afterward, the implementation of generators as the elementary cells will be discussed. Here the focus will too be on the technology-agnostic description but also code reuse will be detailed. Furthermore, the lessons learned from the PCell elementary cells will also be implemented in this iteration.

At least additional generators are introduced which were enabled to be implemented with the XCell framework approach. These will offer further automation besides the elementary cells.

6.4.1 Constraint based Layout Generator Framework XCell

With the skillbridge there is a strong foundation to build a layout generation framework tackling all the in the before section mentioned challenges.

First, the basic concept and interaction between the user will be described. From there the detailed implementation and its most important aspects will be discussed.

In figure 6.11 an interaction diagram on how the interaction between the project author and specific XCells, a user and Virtuoso, and between Virtuoso and the framework takes place. A specific XCell implementation is a constraint-based layout generator that uses the XCell framework. Specific Xcell implementations are grouped into XCell libraries.

Definition 12. The author of these XCell libraries is defined as *project author*.

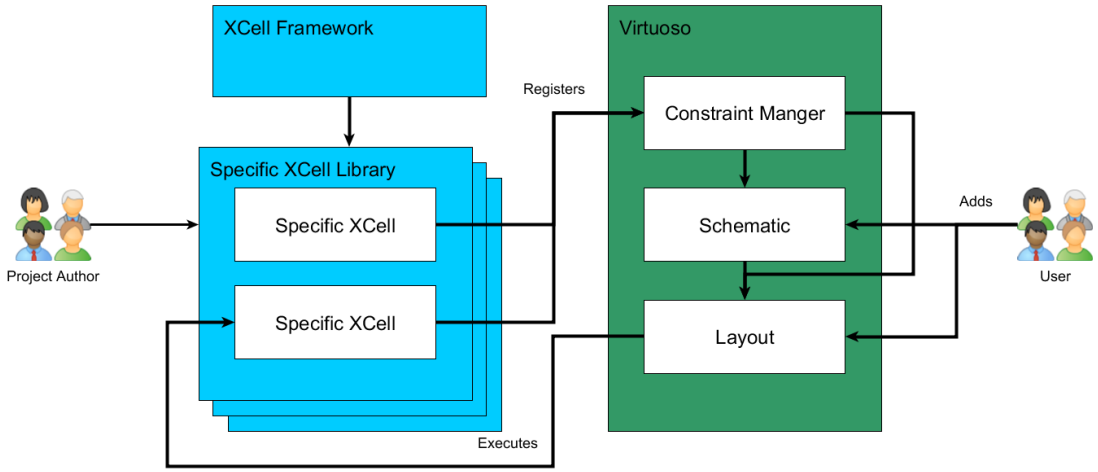


Figure 6.11: Constraint based Layout Generator Interactions modified from [67]

The user also applies constraints to the schematic or layout elements. Different tools offer constraint management and interfaces. In the case of Virtuoso, a custom user constraint interface exists [68] which will be used as one possible implementation in the framework.

Definition 13. The *user* is the creator of schematics and layouts in Virtuoso.

The XCell libraries are registered to the Virtuoso constraint manager which has an interface for custom constraints. The XCell implementations are registered within this interface from the XCell framework.

A typical workflow for the user is to create a schematic with some devices and use the provided custom constraints registered from the XCell library. Constraints from the schematic are transported to the layout when creating the layout view.

In the layout entry, these constraints can be modified and additional constraints can be added. Furthermore, in the layout entry, a GUI is offered to the user to trigger the layout generation of the XCells. XCells cannot only be generated but also regenerated and cleaned. The generated cells will be grouped after generation and all corresponding shapes are marked with their constraint ID. This is an important function for the user to interactively test different constraint parameters or apply updated generator code. Another aspect that can be modified in this menu is the execution order of the constraints which may be interdependent and hierarchical.

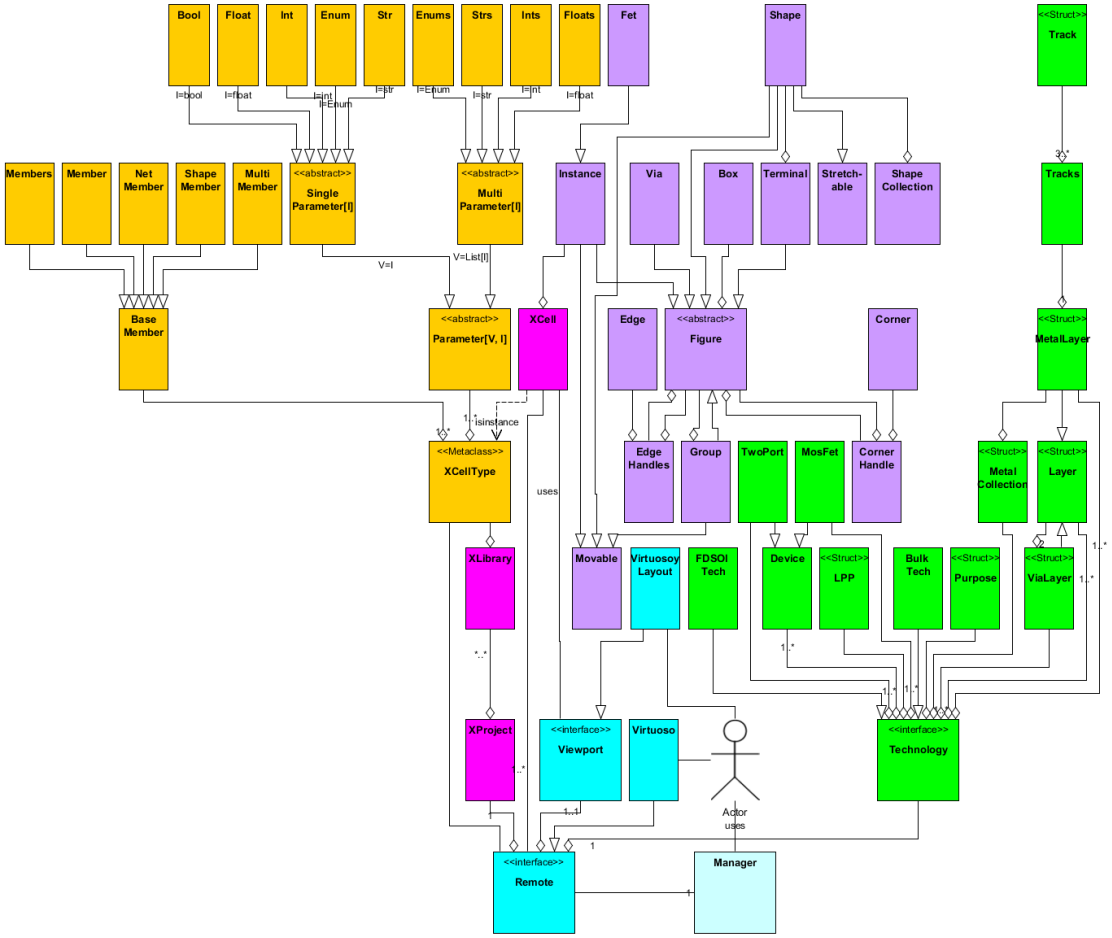


Figure 6.12: Class hierarchy XCell framework, extended from [67]

This constraint-based user interaction and abstraction to create technology-agnostic layout generators is implemented in the XCell framework. For this, the framework offers two main functionalities. First, it provides an infrastructure to communicate between layout tools and generators. Secondly, it provides abstractions to implement constraint-based parametrizable generators. In figure 6.12 the class hierarchy of the XCell framework can be seen.

The class hierarchy shown should only serve as an overview and thus does not give attribute or method details of the classes. Nevertheless, it gives a good top-down overview of the XCell framework. In this figure different corresponding aspects are color-coded.

The entry point of the XCell framework is the Manager. It offers a CLI interface to access the Remote. Via this interface, XCells implementations can be registered and called to generate, clean, and recreate layouts.

The Remote class offers an abstract interface to the ASIC design environment. In this case, the Remote interface is implemented with the Virtuoso class and is composed of the Viewport, a specific technology through the abstract Technology class and the XProject class. In the Remote different interfaces are defined to trigger the generation, clean, and recreate step. Furthermore, constraints can be registered and XProjects are composed. In Virtuoso the concrete menu entries are build up and the Virtuoso-based custom constraints are set up. Also, some additional Track based functions are mapped. This includes menu entries and shortcuts to switch between different tracks and to activate/ deactivate them. Furthermore generated constraints can be snapped to these tracks.
The Viewport is an abstract interface to the layout entry which is implemented with the VirtuosoLayout class. The Viewport offers abstract methods to query and manipulate layout objects as instances, shapes, groups, or vias.

The Technology class composed by the Remote is applied to all generate steps to execute the layout generators on one specific implemented technology. The specific implementation classes act as the technology abstraction and definition. In this example, these classes are BulkTech and FDSOITech. The technology abstraction class hierarchy shown in green will be detailed in the next subsection.

The XProject class is composed of several defined XCell libraries in which the different layout generators are organized. XProject offers some attributes and methods to offer a simple interface to access and add additional libraries or list the XCell generators included. The actual registration and managing of the libraries are implemented in the XLibrary class. XCell class is an instance of XCellType. The XCell class implements the base class of every implemented generator. Its main task is to offer the project author a comfortable interface to write generators. For this, it passes the Viewport and the corresponding technology to the project author. Detailed examples on how the XCell class is used to implement generators will follow within this chapter. For one a minimal generator example will be described to introduce and show the main concepts. After the minimal example, the implementation of the elementary cells and some additional generators will be detailed.

An implemented XCell will almost always be composed of instances. These Instances are modeled with the Instance class. Furthermore, the access and interoperation of instances, groups, shapes, and boxes are implemented with the classes marked in violet. The abstract class figure is either implemented by an Instance, a Terminal, a Via, a Shape, or a Group. A Group is composed of multiple Figures. The Figure is furthermore composed of CornerHandles and EdgeHandles which adds properties to access edge values and corner coordinates of the corresponding figure. Additionally, the Box, Shape, and ShapeCollection classes offer an interface to create or query geometry objects. The Stretchable or Movable class adds the according functionality to the corresponding Figure implementation. Most of the noted classes are used to modify or query existing objects in the layout. An exception is the Box class which can be used to build up geometry which then can be drawn with the implemented Viewport. This is mostly used when writing generators or methods to create specific parametrizable shape constructs. Detailed examples will follow within this chapter.

XCellType is an instance of XCell and implements the base class variables. XCell implements the behavior of these. From XCell the project author derives the layout generators. Additionally in the implemented generators, BaseMembers and Parameters can be composed. In the generator, this can be done as a simple class variable assignment, while in the background these parameters or members are prepared for the Remote to be added to the corresponding custom constraints.

Constraints are applied to so-called members. Members are schematic or layout instances. To describe which Members are allowed for a constraint the following classes exist.
A BaseMember of a layout generator can be Member which represents a single instance on which the corresponding constrain can be applied, Members represent multiple allowed instances and MultiMember represents multiple instances of a single instance type. Additionally, constraints can be set on nets or shapes which are represented by the Net and ShapeMember classes.

6.4.2 Technology Abstraction

To achieve technology-agnostic generators a technology-independent interface was implemented with the class XTechnology. It implements two main ideas. First, it offers an abstract interface with which a target technology can be mapped with a corresponding class implementing XTechnology as seen in the example below. Secondly, XTechnology

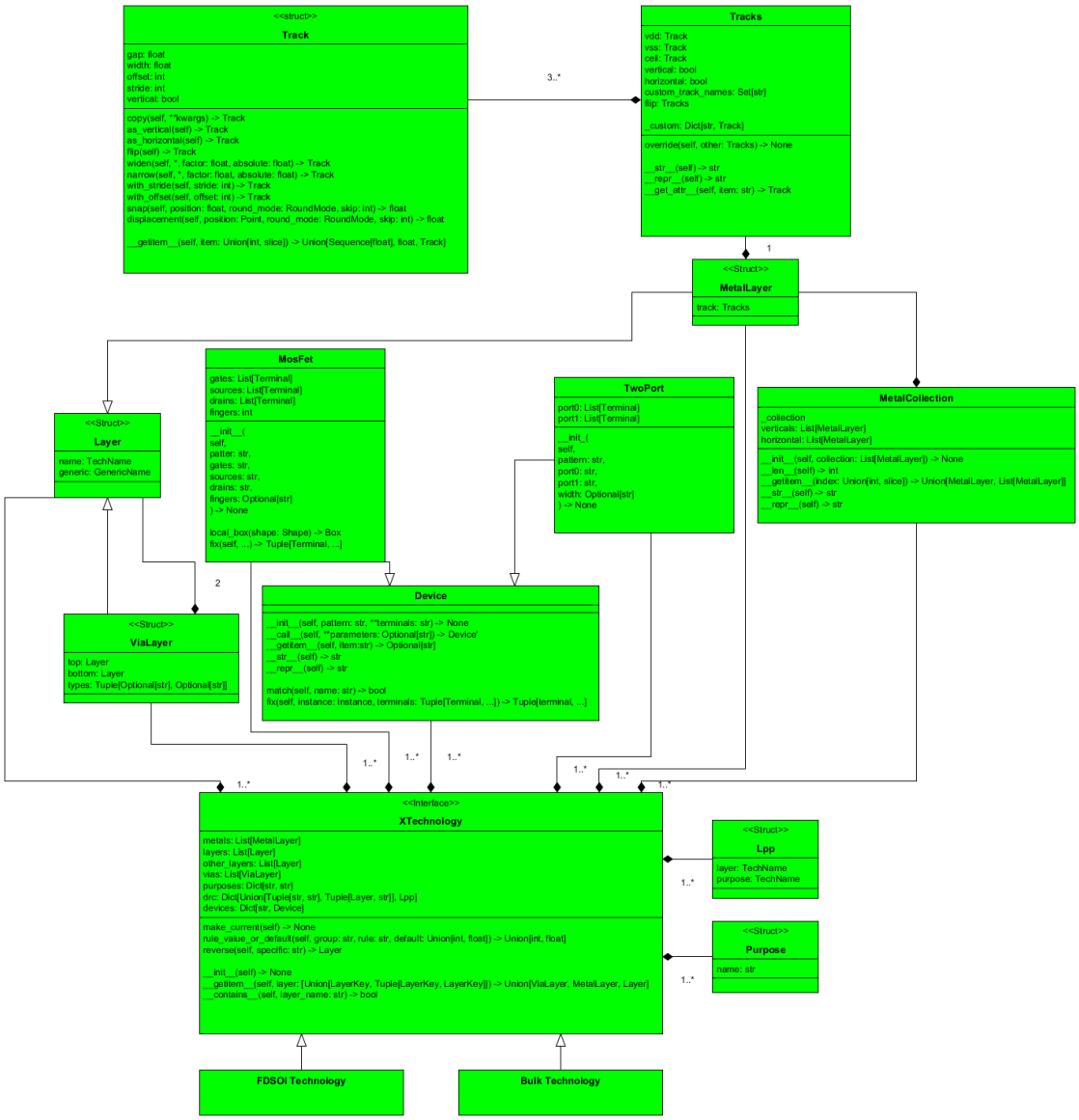


Figure 6.13: Class hierarchy XTechnology

additionally offers powerful methods to access these specifics.

In figure 6.13 the class hierarchy of XTechnology is shown. Only public methods are noted since they are sufficient to understand the core functionality and keep the figure clearly represented. XTechnology is an interface that has to be realized by a concrete Technology class in our example TargetTech which inherits from XTechnology in line 1. In the class hierarchy, XTechnology is realized by FDSOITechnology and BULKTechnol-

ogy. For these XTechnology offers to map technology specifics to generics with building up a specific composition of Devices, Layer, MetalLayer, ViaLayer, MetalCollections, Purpose and LPP Objects.

```
1  class TargetTech(XTechnology):
2    active = 'RX'
3    poly = 'PO'
4
5    m1 = 'M1', fine_tracks
6    m2 = 'M2', fine_tracks.flip()
7    m3 = 'M3', coarse_tracks
8
9    local_route = [m1,m2]
10   mid_route = [m3]
11
12   via_m2_m1 = 'V1', ('Vx', 'VxBAR')
13   via_m3_m2 = 'V2', ('Vx', 'VxBAR')
14
15   drc_m1_exclude = 'M1', 'exclude'
16   drc_m2_exclude = 'M2', 'exclude'
17   lvs_diode = 'DIODE', 'lvs'
18
19   fet =MosFet('*fet*', gates='g*', sources='s*', drains='d*', fingers='nf')
```

To comfortably define the mappings for each object within the composition XTechnology will take all given class variables in its __init__ method and build the corresponding attributes and compositions on object creation. Depending on how the class variable was defined a different object will be composed as example lines 2-3 will result in a layer mapping.

```
1 tech = TargetTech()
2 tech.poly
3 >> Layer(name='RX', generic='active')
4 tech.poly.generic
5 >> 'active'
```

In the above listing, an object is created from our defined TargetTech, and with this, the layer can be accessed via a class property returning the Layer object which represents a

struct with its attributes generic and name. The name is the specific technology name while generic is the generic name the layer gets mapped to. The project author works with the generic layers or directly with the defined layer objects offered by the implemented XTechnology interface.
MetalLayers are created when assigning a specific name and corresponding Tracks objects. This can be seen in lines 5-7 from our TargetTech example. The MetalLayer struct inherits both properties from Layer and adds the track property which is of type Tracks. The Tracks are further composed of multiple Track objects. An example of a Tracks definition can be seen in the example below. Here two Tracks are defined one resulting in finer tracks for lower metals and one for coarser tracks for upper metals. Since each Tracks setup will be added to a MetalLayer the Tracks definition has a common vertical or respectively horizontal property. Each Tracks setup consist of multiple Track definitions.
In this example definition, the VDD and VSS Tracks are defined to alternate with a stride of two and a corresponding offset. The width defines the default metal width on these Tracks and the gap the distance between two Tracks. A horizontal or vertical attribute can be set as shown in line 5 and 11.

```
1 fine_tracks = Tracks(
2   vdd=Track(width=0.2, gap=0.25, offset=0, stride=2)
3   vss=Track(width=0.2, gap=0.25, offset=1, stride=2)
4   cell=Track(width=0.1, gap=0.125, offset=0, stride=1)
5   vertical=True
6 )
7 coarse_tracks = Tracks(
8   vdd=Track(width=0.3, gap=1, offset=0, stride=2)
9   vss=Track(width=0.3, gap=1, offset=1, stride=2)
10  cell=Track(width=0.2, gap=0.25, offset=0, stride=1)
11  vertical=True
12 )
```

MetalLayer can be accessed as shown in the following example from a MetalLayer the corresponding track for a specific purpose can be selected and in the example, the gap can be read. In the class hierarchy from 6.13 it can be seen that the Tracks class implements the __get_attr__ method. With defined custom tracks can too be accessed via a property, where the property name is dynamically built from the given custom track name. With the __getitem__ method Track behaves as a sequence implementing self[key].

This behavior can be seen in the example from lines 5-10. In line 5 a simple access is done and the 3rd track is read out which yields 0.375 since the cell track is defined with a gap of 0.125 and a stride and offset of one. The sequence can not only be accessed with an integer but also a slice as shown in line 8 where the tracks from 0-3 are taken. In line 10 it is shown that also negative Track indices may be used. If a slice definition does not result in a closed list a Track copy with the corresponding offset and stride is returned as shown in line 12.

```
1  tech.m1
2  >> MetalLayer(name='M1', generic='m1', tracks=Tracks(...))
3  tech.m1.tracks.cell.gap
4  >> 0.125
5  tech.m1.tracks.cell[3]
6  >> 0.375
7  tech.m1.tracks.cell[0:3]
8  >> [0.125, 0.25, 0.375]
9  tech.m1.tracks.cell[-2]
10 >> [-0.125]
11 tech.m1.track.cell[::3]
12 >> Track(gap=0.125, width=0.2, offset=0, stride=3)
```

Additionally, the Track class also offers methods to derive Track copies. The methods self.as_horizontal and self.as_vertical can be used to create a Track copy with the corresponding orientation. self.narrow and self.widen are used to create a copy with a changed width. self.with_stride and self.with_offset are used to get a copy with changed stride or offset. The self.copy method can be used to create a copy with arbitrarily changed parameters.

With lines 9-10 in our TargetTech class, we define a MetalCollection. A metal collection is a group of MetalLayers. In most technologies, some metals can be grouped for their uses. Often M1 is only used for local route inside and near devices. Additionally, it has become more common that M1 and M2 are doubled patterned and thus both used at local device level. The next metals are often used on block level and the upper metals for the system level interconnections between blocks. This grouping is represented by the MetalCollections class if in TargetTech a list of metals is assigned to a value a MetalCollection is composed and can be accessed as in the following example. MetalCollection behaves like a list as it implements __getitem__ and __len__. Furthermore, with

the verticals and horizontals property, the MetalCollection can be filtered accordingly and returns a List of MetalLayer as seen in line 4.

```
1 tech.local_route
2 >> MetalCollection(MetalLayer(generic='m1', ...), ...)
3 tech.local_route.horizontals
4 >>  [MetalLayer(generic='m2', ...)]
```

In lines 12-13, of the TargetTech class vias are assigned with given the technology-specific Via name and a tuple of the via types available in the technology. On object creation, the corresponding ViaLayer objects are created which inherits from Layer and is composed of 2 Layer or MetalLayer. For this, the generic name is important since from them the layer pair is build up. With the syntax via_{upper_layer}_{lower_layer}.

```
1 tech.via_m2_m1
2 >> ViaLayer(...)
3 tech['m1']
4 >> MetalLayer(...)
5 tech[tech.m1]
6 >> MetalLayer(...)
7 tech[tech.m2, tech.m1]
8 >> ViaLayer(...)
```

In the above listing, it is shown how to access vias in line 1 with the created property. Additionally to the property access to layers the XTechnology class also implements __getitem__. With this, all layers including MetalLayer and ViaLayer can be accessed line 3-8. Keys may be strings of the generic layer names or the Layer object. If two Layer objects or generic names are given the corresponding via object is returned.

In line 15-17 of TargetTech LVS and DRC layer purpose pairs are mapped these are often needed for in example for the fill layers used in the layout entry.

In the last step, the devices are mapped. In our example, a MOSFET gets mapped and given match strings which determine which devices will be mapped as MOSFET and how their terminals and parameters are matched. It should be noted that the match string may map multiple devices. This is especially useful when a technology organizes

different types of the same devices in different cellviews.
Device objects are not directly used through the technology mapping but rather indirect as defined Members in a Layout generator implemented with XCell.

On object creation, the XTechnology class runs a check if the most important technology aspects are mapped.

6.4.3 Layout Generators

With the main functions described a layout generator can be implemented. This subsection showcases a minimal layout generator to introduce the main concepts based on the above-described framework.

The example below implements an XCell for a meandering resistor here called snake resistor. For this, the SnakeRes class which represents the generator inherits from XCell. Some variables and methods offered from the base class XCell have to be implemented.

With title and menu the subcategory of which the resulting custom constraint will be found in the Virtuoso constraint manager and the custom constraint name will be set, line 2-3.

With the phase variable, the generation phase is set. It is possible to build hierarchically dependent layout constraints especially when grouping and placing elements. For this, the phases exist the PrePhase is a defined phase where each layout generator is independent, line 5. PostPhase is a defined phase where dependencies between constraints can be organized over a menu. Constraints in the PrePhase phase are always executed before the PostPhase constraints.

Afterward, constraints members can be defined. This determines on which device or devices the constraint can be applied. In our example, in line 7 we define a single device as a target which should be mapped as an LVS resistor of the type TwoPort.

```
class SnakeRes(XCell):
  title = "Snake Resistor"
  menu = ["Passive"]

  phase = XCell.Phase.PrePosition

  device = Member("lvs_res", TwoPort)

  turns = Int("Number of Turns", min=1, max=100, default=10)
  stride = Int("Stride on the cell Track", min=1, max=10, default=1)
  seg_lengths = Float("Segment length in um", default=0.5)

  def draw(self, viewport: Viewport, tech: XTechnology) -> None:
    port_shape = self.device.port0.shapes[0]
    layer = [metal for metal in tech.metals
        if port_shape.ptr.layer_name == metal.name][0]

    track = layer.track.cell.as_vertical.with_stride(self.stride)
    self.device.snap(track, mode=Track.left, ref=port.shape)
    self.device.save_track(track, ref=port.shape)

    box = Box.union(*self.device.port0)
    box.top += self.seg_lengths
    path = viewport.path_from_box(layer, box)
    top_shape = path.box.top-path.box.width/2
    bottom_shape =  path.box.bottom+path.box.width/2

    for n, horizontal in zip(
      range(self.turns-1),
      cycle([top_shape, bottom_shape]),
    ):
        start = (path.box.left, horizontal)
        box = moved_copy(path, (track.offset+track.gap*track.stride, 0))
        end = (box.right, horizontal)
        viewport.path_from_ends(layer, start, end, box.width)
        path = viewport.path_from_box(layer, box)
```

The constraint parameters can be defined with the Types offered by the abstract parameter class in the background these parameters will be added to the corresponding Virtuoso Custom Constraint.

In this example, the resistor will be drawn onto the cell track of the corresponding metal layer. Constraint parameters are a stride for the tracks, a segment length, and the number of segments lines 9-11. For the Parameter default, min and max values can be defined.

The draw method implements the code which is executed on generation. It provides the technology interface as *tech* of type XTechnology and the *viewport* from type Viewport, which offers a way to access functions for the layout entry.
In lines 14-15 the port0 shape is selected from the TwoPort and assigned to *port_shape*. Afterward, the layer of the port shape is read out and assigned to the layer.
In lines 17-18 the TwoPort is snapped to the vertical cell track of the read-out layer. The port0 shape will be used as the reference for the user to snap the resulting group to the tracks. With the save track command this track is set as a reference when the constraint finishes generation. The resulting group can now be snapped using a defined keyboard shortcut or from the XCell menu entry.
In lines 22-26 the box of port0 from the LVS resistor on which the constraint is set is queried. Afterward, the top edge of the Box is extended by the segment length resulting in our first segment. The initial path is drawn from this Box in line 24. The path_from_box method takes the Layer and Box, draws the path, and returns the proxy object of type Shape. The top shape edge position and the bottom shape edge are determined as reference shape for the resistor connections between each segment line 25-26.
From lines 28-36 the additional segments are drawn within the for loop. In the loop, a counter variable n is introduced and combined with a horizontal shape which alternates between the top shape reference and the bottom shape reference. From this, a start and endpoint are calculated to draw the horizontal connection with path_from_ends in line 35. Afterward, the next segment is drawn from a copied box which is moved based on the track offset, gap, and stride.

In figure 6.14 the resulting custom constraints are applied to each LVS resistor in a schematic. On right left side of the schematic entry and layout entry, the constraint editor can be seen. Via the constraint editor, the constraints can be set in schematic and layout. The created custom constraints can be found under the custom constraint menu in the subfolder Passives asset in the SnakeRes class. The active constraint on the schematic and layout can also be seen in this window. Snake Resistor corresponds to our set title. When selecting the constraints the parameters can be configured. For LVS resistor R3 this is 5 turns, a stride of 1, and a segment length of 2um. In the layout, the

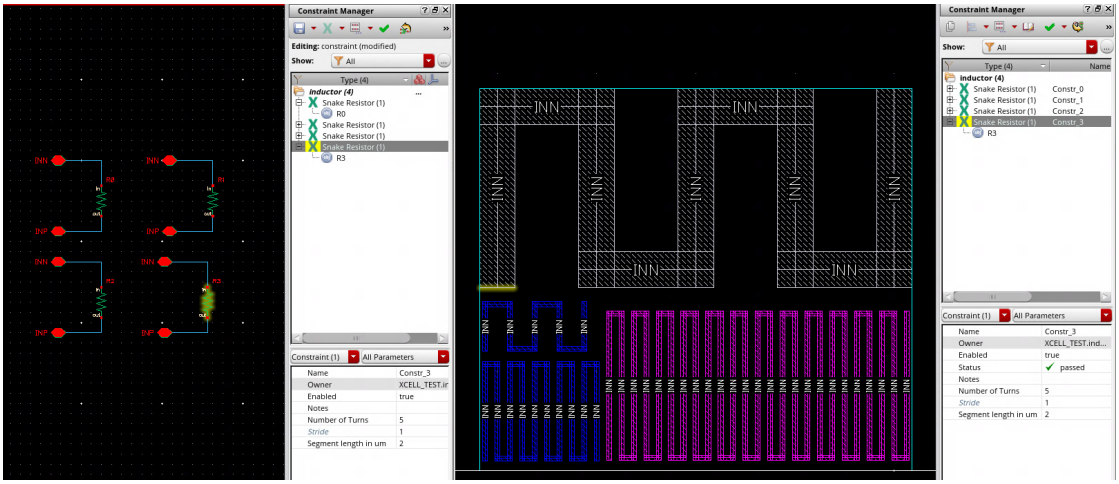


Figure 6.14: Example Snake Resistor custom constraint XCell

result of generating the constraints in the layout can be seen.

Some aspects can already be noted in this minimal example the approach of constraining existing devices in this case the LVS resistor allows to inherit several of its parameters. In this generator, the layout generation inherits the width and the layer of the LVS resistor. Another important fact shown with this example is that other PDKs might organize their devices differently. While in our FDSOI PDK the layer of the LVS resistor is included in the CDF parameters of the vendor PCell in the used bulk PDK for each metal layer a separate PCell for this individual layer exists. The naming scheme in this PDK is rm1 to rm9 for the different metal resistors. This is already covered by our layout generator since the layer is extracted from the underlying shape of the LVS resistor and not from the CDF parameters. For this, the TwoPort has to be correctly mapped for both technologies. The differences in the mappings are shown below as seen the specific port names and parameters are mapped to the generic port and parameter names.

```
## LVS resistor mapped in the FDSOI PDK
lvs_res = TwoPort('lvsres', port0='in', port1='out', width='w')
## LVS resistor mapped in the bulk PDK
lvs_res = TwoPort('rm*', port0='PLUS', port1='MINUS', width='w')
```

6.5 Elementary Cells Xcell Implementation

From the initial elementary cell implementation, several lessons learned were taken. Several of these were limits set by the PCell Designer as code reuse and technology-agnostic approaches. But there are other more layout-centric lessons learned.

Between the PCell elementary cell implementations and the XCell framework development, an updated PDK was released for our FDSOI technology. In this iteration, the EM rules were updated and allowed in certain cases multiples of the maximum allowed current for shapes since the initial EM deck was very conservative and reworked. While with the old PDK the clip variants of the elementary cells were the default in the new PDK version this shifted towards the traditional stripe setup which also will result in more compact designs.

A disadvantage of the initial elementary cell implementation was that the guardrings could be drawn around each cell depending on the configuration. Using this feature often resulted in larger than necessary layouts and less overall M1 and poly density since these individual guardrings were not abutable and had a larger keepout area. In this approach guard rings are not included in the elementary cells but rather implemented as additional layout generators.

These lessons will be implemented in the XCell framework-based approach.
The second main focus is code reuse. The XCell implementations are not artificially limited by a given inheritance structure or similar so all Python language concepts can be used and thus an object-oriented approach is possible and was chosen to implement the elementary cell generators. The implemented generators are a PassiveLoad, a TransistorPair, a Transistor Array, and a CMOS constraint. Besides these cells, an NRing, PRing, and SubRing generator are implemented and a PostCMOS generator.

6.5.1 Pre Phase Elementary Cells

In figure 6.15 the class hierarchy of the implemented elementary cells can be seen. For transistor generators abutting devices in a row is always needed. To comfortably allow the user to create abutable rows of transistor devices, helper objects are introduced.

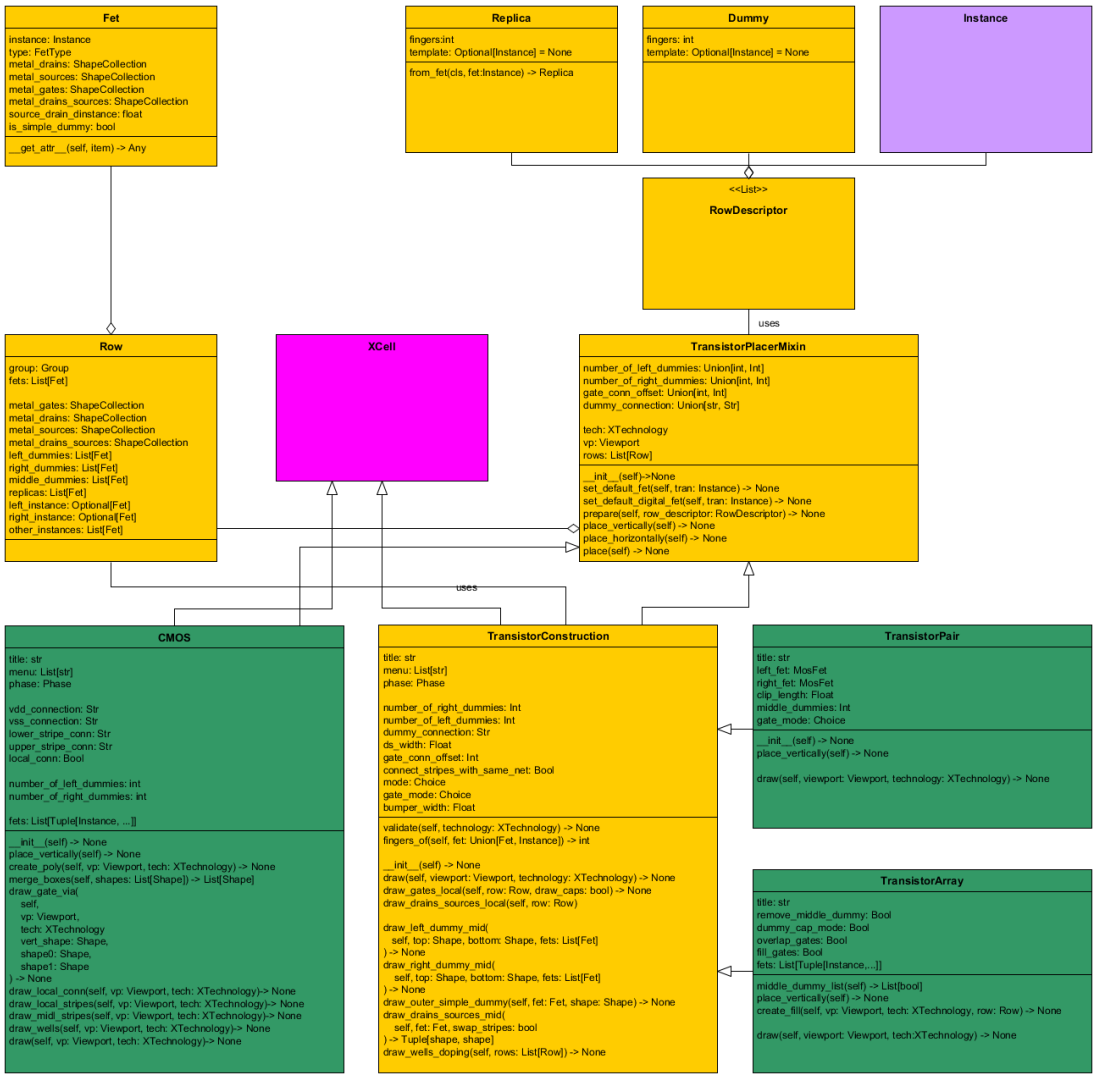


Figure 6.15: Class hierarchy elementary cell generators

To describe an abutable row a RowDescriptor can be used which is a List containing elements of the type Replica, Dummy, and Instance. The Instance is typically the defined transistor constraint member and thus the transistors the constraint targets for layout generation.

Dummy instances mark copies of the members where a parameter will be changed. For dummy transistors, this parameter is the fingers. Dummies will be connected to an in the constraint defined dummy net.

The Replica class is used to mark full dummies of the instances meaning without pa-

rameter change but also connected to a dummy net.

The TransistorPlacerMixin is a class encapsulating all needed methods and behavior to place and abut transistors. For this, the RowDescriptor is used.
Notable is the prepare method. The prepare method takes the RowDescriptor and creates the described replicas and dummies. Now that every instance exists the prepare method abuts all devices with the Viewport.chain() method. This returns a Group of the layout instances. This Group represents the generated Row and will be wrapped in the Row object which is appended to self.rows thus with multiple calls of prepare, multiple rows can be added. The TransistorPlacerMixin further offers a method to place the rows and to set default transistors, which will set transistor parameters to parameter which will fit perquisites for the generation and best practice.

Below an example of how the prepare method is used in the transistor pair on the generator. For the transistor pair, the right transistor is mirrored line 1 afterwards the row is prepared and then the place method from the inherited TransistorPlacerMixin is called.

```
1  self.right_fet.mirror_horizontal()
2
3  self.prepare([
4      Dummy(self.number_of_left_dummies, self.left_fet),
5      self.left_fet,
6      Dummy(self.middle_dummies, self.left_fet),
7      self.right_fet,
8      Dummy(self.number_of_right_dummies, self.right_fet),
9    ])
10
11 self.place()
```

In figure 6.16 two examples are shown on the left two PMOS transistors with the TransistorPair constraint and on the right, a TransistorArray constraint set on an NMOS transistor with the multiplier of two for both in the middle the generated intermediate result of prepare is shown and in the end the full layout. Both target transistors have minimal gate length and finger width of 490 nm. The constraint parameters are set as follows, for the TransistorPair the number of dummies for left, right dummies are set to one and the middle dummies to four. For the TransistorArray number of left dummies

and the number of right dummies was chosen to be four. A more detailed description of the TransitorPair and TransistorArray and its constraint parameters will follow.

The Row class implements a wrapper around a layout group of transistor instances. The Row class is composed of a wrapper of the transistor instances the Fet class. Both Row and Fet class offer some properties to simply access metal terminals. The Row additionally offers properties to access specific transistor instances or dummies. One or multiple Row objects are composed by the TransistorPlacerMixin from a RowDescriptor which is given to the prepare method.

The TransistorConstruction class is the base class for the TransistorPair and TransistorArray generators. TransistorConstruction inherits from the TransistorPlacerMixin and XCell. TransistorConstruction offers the base methods and properties needed to create a fanout structure typically used in analog transistors. Additionally, TransistorConstruction also defines common device parameters for both generators as the number

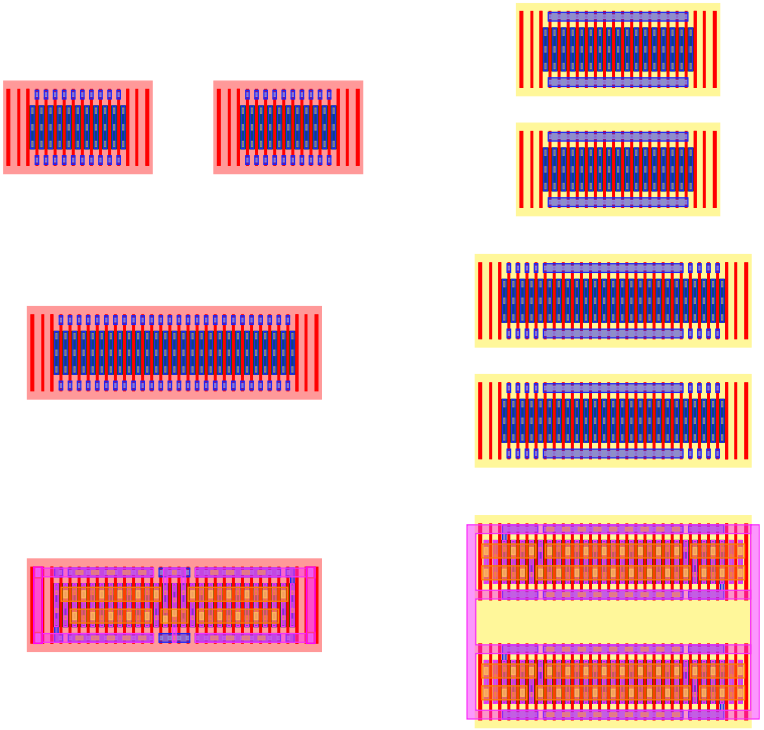


Figure 6.16: Left: TransistorPair constraint from the constrained devices to the generated layout, right TransistorArray constraint

of left and right dummies, the dummy connection net, gate connection modes, and general connection modes, and more. It also defines the common phase in which these generators are run and a common sub-menu.
The implemented common draw methods are divided into draw methods are organized into two categories. Function and layer group which accomplish different tasks as to draw the dummy connections or the main stripes and additionally into draw local and draw mid. This refers to the layers in the corresponding MetalCollection defined in the implemented technology. This was chosen to get a better overview and better maintainability of the code. Especially when an issue with the generators arises potential bugs can be identified fast.

The TransistorPair and TransistorArray generators are the most versatile elementary cells, as they can be used for differential pair structures, passive loads, current sources or even CMOS circuits with a high driving strength. Both generators inherit from TransistorConstruction. The only difference in both generators is how the abutted rows are prepared. For the TransistorPair the right transistor member will automatically be mirrored to result in a symmetrical layout. This was shown in the above code listing. The TransistorArray can prepare and create multiple rows as shown in figure 6.16 on the right a 1x2 transistor array was generated. The TransistorPair constraint can be applied to exactly two transistors. Indifference the TransistorArray can be applied to multiple transistor instances. If the TransistorArray constraint is applied to multiple transistor instances each instance will result in a new column. Indifference the multipliers of the transistors result in multiple rows. Depending on the constraint transistor members it is often the case that it will not result in a full matrix. Empty places will be filled up with replica dummies. The generated layout will thus always be a full array.
While multiple parameters of the TransistorArray and TransistorPair are inherently used from the constraint members, as finger width, the number of fingers, multiplier, transistor type additional constraint parameters exist.
Some of the Parameters are shared through the TransistorConstruction class additionally the TransistorPair has optionally middle dummies which may be added and when the clip mode is chosen a clip width. The TransistorArray additionally adds a dummy capacity mode which connects the added dummies as capacitances which are especially for big current sources useful. The TransistorArray automatically adds a dummy transistor between each column. With the remove_middle_dummy option, the constraint checks the connectivity of neighboring devices and removes the middle dummy if possible.

In figure 6.17 some example configuration of TransistorPair and TransistorArray constraints are shown.
In the left half, multiple TransistorArray constraints were set in a) for a 490nm minimal length NMOS transistor with a multiplier of 2 and several left and right dummies. In b) a bigger Array is shown with 150nm length 990nm finger width 3 instance members with a multiplier of three two and one. Resulting in multiple Replica devices filling up the array. c) shows a TransistorArray containing a single device while in d) 2 devices with a multiplier of 2 are shown. In d) the constraint parameter is additionally set to not draw metal stripes onto the dummies.
On the right half, TransistorPair constraints are shown. Below e) and f) are a TransistorPair constraint on 490nm finger width transistors. e) without middle dummies where the constraint checks if both pairs can be abutted and f) with multiple middle dummies. In these examples, all TransistorPair constraints are applied on minimal length transistors. The dummies are always constructed in the correct way avoiding coloring errors as odd cycle violations. In g) a 990nm finger with variant is shown with single finger dummies and multiple middle dummies. In h) a small TransistorPair variant is shown where the left and right dummy are asymmetrical with 1 and 2 fingers without dummies and stripes is shown. i) shows a 990nm finger width variant without stripes. In j) a variant for 990nm finger width is shown with multiple left and right dummies where the dummy connection is done over a jumper via to avoid coloring violations

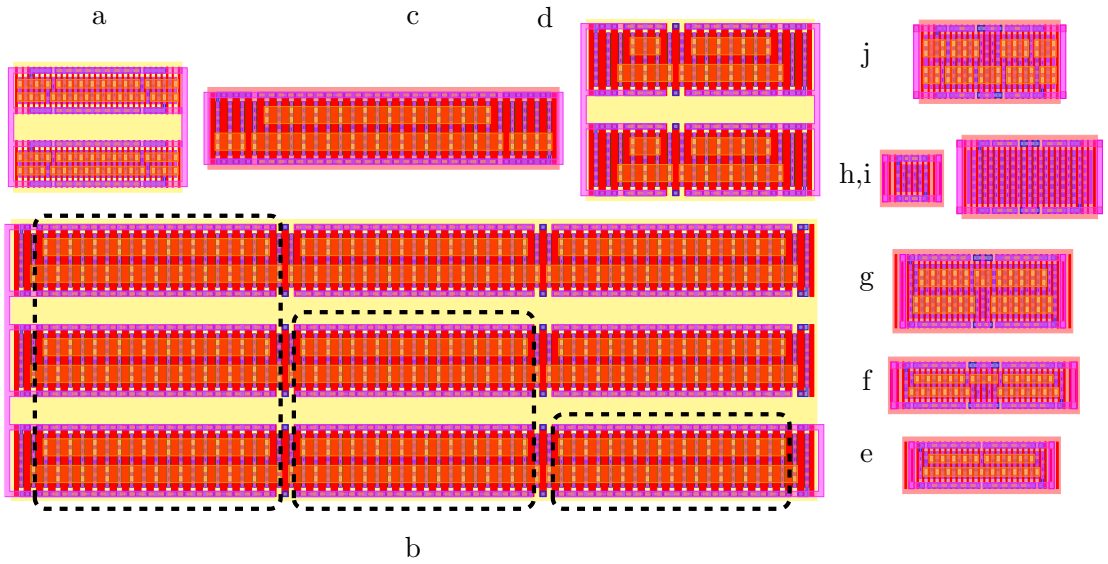


Figure 6.17: Left: TransistorArray examples, right TransistorPair examples

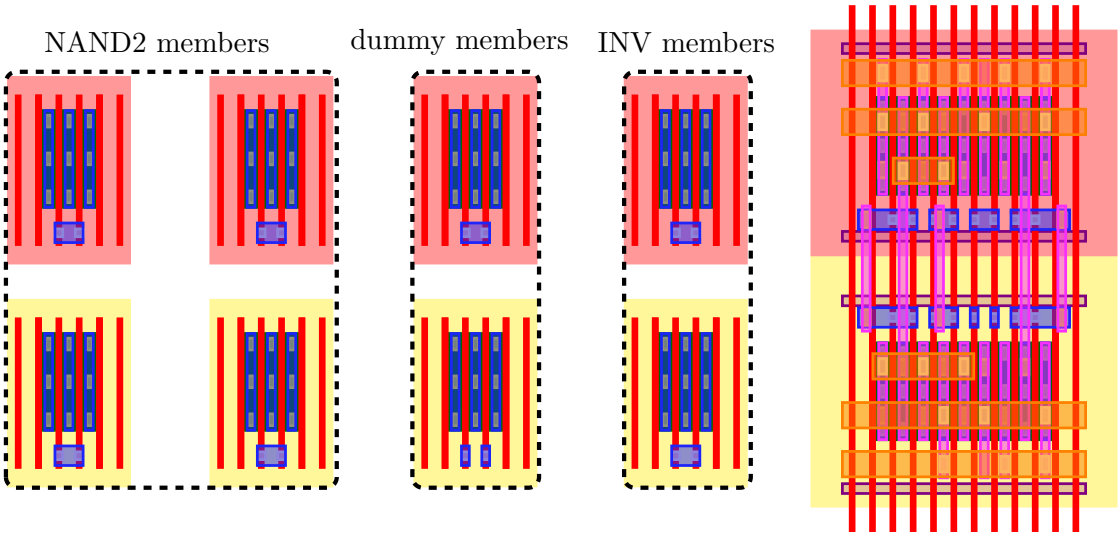


Figure 6.18: Left: Member NMOS and PMOS transistors on which the constraint is applied, right: generated layout

Another implemented generator is implemented with the CMOS class which inherits directly from XCell and the TransistorPlacerMixin. The CMOS generator offers the user a constraint that can be applied to the same number of NMOS and PMOS transistors. On generation, the transistors are abutted and placed in two rows using the methods from TransistorPlacerMixin. The generated CMOS cell will create VDD and VSS rails within a defined height depending on track constraint parameters. An example of a fast NOR can be seen in figure 6.18. On the left, the member transistor pre-generation is shown the single finger devices on the left and right are dummy transistors to encapsulate the active transistors in the middle.
Further constraint parameters offered by the CMOS generator are the connection names that will be applied on the rails and the matching transistor connections will then be routed to the rails. The default values are VDD and VSS. Furthermore, the connections of the upper stripe and lower stripe interconnecting an internal CMOS net can be set. Furthermore, a constraint parameter exists to switch if simple internal signal connections should be routed. These internal connections between NMOS and PMOS rows are generated simply by looking for neighboring pairs and connecting them with an m2 stripe on the cell grid and matching m1 extends. This simple routing is only helpful in simple CMOS cells as simple combinatoric blocks (NAND, NOR, AND, OR, or XOR) but is limited for more complex blocks as in example latches or flip flop structures.

The CMOS structures are generated to be abutted with other CMOS blocks resulting in a very dense layout.

The Pre Phase generators are on their own not fully DRC or LVS clean. For the TransistorPair and TransistorArray well-taps or well-rings have to be added. In the case of the CMOS cells not only well-taps have to be added but also in depending on the gate length a special stop dummy structure has to have a specific distance and length. To solve this issue the corresponding post-phase constraints are implemented.

6.5.2 Post Phase Elementary Cells

For the XCell generations, post-phase cells may be interdependent and accordingly a dependency between these cells can be defined. In this subsection, an overview of the implemented post cells is given and afterward, some hierarchically defined examples are shown.

In figure 6.19 the class hierarchy of the post cells is shown. The GuardRingMixin class offers a method to draw tap segments of a specific well type for example N-Type, P-Type, and Sub-Type. While in a bulk technology only N-Type and P-Type are available in a FDSOI technology all three exist and will additionally set or generate the correct triple well and hybrid layer. This tap is created in the form of a single path segment.
The GuardRingBase class inherits from XCell and GuardRingMixin. GuardRingBase offers all common constraint parameters for the different type-specific guard ring generators. Guardring base sets a default title and a common menu entry for the Guardring constraints. For all guardrings constraints, the constraint members are of type Members meaning allowing an arbitrary number of constraint members of an arbitrary type. Shared constraint parameters are exclude-layers that can be chosen and will be applied to the constraint area. Furthermore constraint parameter for each side exits which allows selecting if and what metal layers should be generated for this side. Only when M1 has been selected the connection to the substrate is drawn below. Furthermore, each side can offset by a defined number of cell tracks. In the default case, the Guardrings are generated with minimal allowed distance to the constraint devices. Additional parameters are the fill parameter which determines if the area inside the guardring will be filled. If fill is true the correct well layers are drawn, if set to false no well and doping layers are drawn inside the constraint area.
The Boolean constraint parameters active_width and active_length can be set to relate the width or length according to the active layer of the targeted constraint. This is

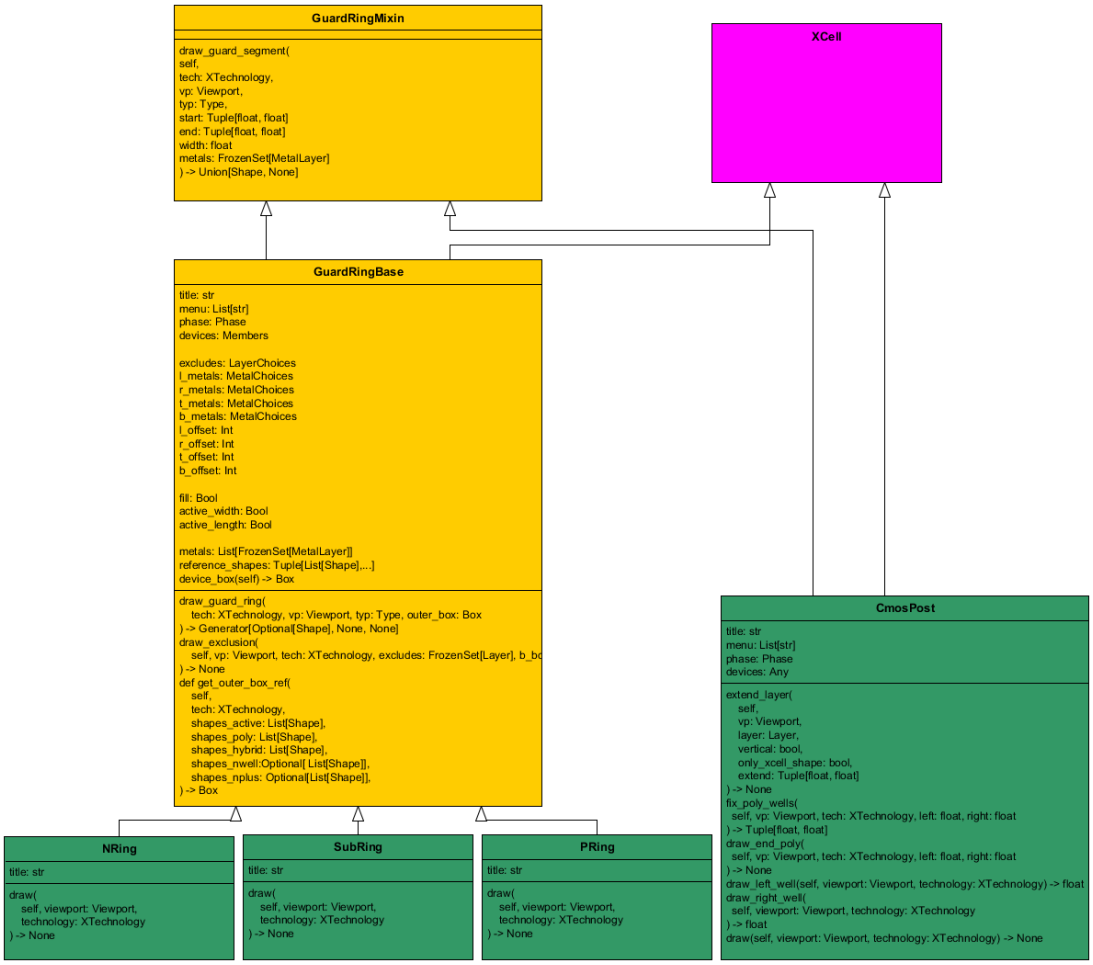


Figure 6.19: Class hierarchy post cells

especially used when minimal well-taps instead of full rings should be generated.

The GuardRingBase offers properties for reference shapes and the constraint device box. The most important functionalities are the offered methods to draw the guardring, draw exclusions or set the reference box.

The NRing, PRing, and SubRing inherit from GuardRingBase and implement their title and implement their draw method which selects the type and draws additional doping layer to realize the chosen type.

In figure 6.20 an example usage of the GuardRings is shown. In this case, a differential

amplifier with an active PMOS load is constraint. The PMOS load P1 and P0 are constraint with a TransistorPair, as the differential NMOS pair N0 and N1. N2 is constraint as a TransistorMatrix. Both PMOS transistors are additionally constrained for a PRing. All devices have a constraint for the NRing and the SubRing.
For this example, the difference in generation for pre and post-phases has to be understood. The pre-phase constraints in this example are the TransistorPair constraint applied on two PMOS transistors and a TransistorArray constraint for the NMOS devices below. Pre-phase constraints are treated independently on generation. The user can either generate the constraints in different ways over the GUI. Only pre-constraints can be generated or only post-constraints if all pre-constraints are generated. For the clean or regenerate commands also both variants exist. In the pre-phase, the TransistorPair and TransistorArray are generated. In this phase, the order of generation for each constraint is irrelevant since they are independent.
The guard ring constraints are post-phase constraints. Post-phase constraints may be dependent on each other and thus dependencies can be defined. These dependencies will determine the generation order. In the shown example the constraint with the highest priority is the PRing, afterward the NRing, and at least the SubRing. This results in the figure shown layout, where the PRring is drawn around its PMOS members. next

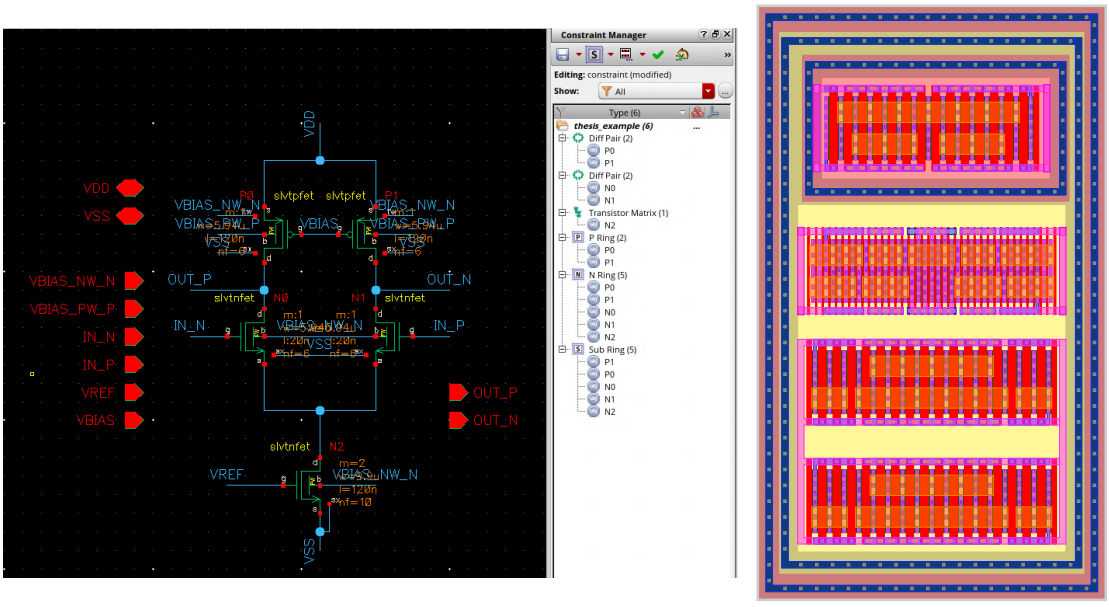


Figure 6.20: Left: Example of a constraint setup for a differential amplifier including pre and post constraints, right: generated layout from the constraints

in generation order is the NRing which is drawn around its PMOS members and NMOS members. The guardring constraint always draws its guard ring around the outer group of its members thus drawing the NRing around the PRing and the NMOS members. The same is true for the SubRing which is drawn last around the NRing.

The generated Rings set their vias on the cell tracks of the local metal layers. If this distance is below the minimal allowed distance the vias are set on 2 times the cell gap. This on the one hand leads to a more uniform layout and additionally allows the user to abut different cells guard rings onto each other resulting in denser layouts.

The CMOSPost class implements a generator that can be applied on multiple CMOS instances generated by the CMOS constraint. The CMOS constraint is abutable everything inside of the generated constraint is DRC clean but only when the transistors are terminated with the right poly stop dummies and with the missing well taps. These additional layout elements are created by the CMOSPost constraint.
For this CMOSPost inherits from the GuardRingMixin and with it all methods needed to create well taps. Additionally, the CMOSPost constraint inherits from XCell. CMOSPost adds a title and menu entry for the constraint. Its constraint members can either be multiple groups of transistors on which the CMOS constraint was applied or multiple instances in which a CMOS device was build up using the CMOS constraint.
CMOSPost implements methods to make the underlying CMOS structure DRC clean. This is done in several steps in the implemented draw method. First, fix_poly_wells is called which fixes the above and below well areas, since originally they are too short and do not meet the in the DRC rules defined overlap. Secondly, with draw_end_poly the stop poly dummies are drawn with the correct distance and width. Lastly left and right well contacts are drawn.

In figure 6.21 an example is shown here 5 NAND instances which were abuttet are the constraint members. The NAND instance uses the CMOS constraint in its corresponding layout view. These are shown pre-generation on the left and after generation on the right.

6.6 Testing and Deployment

One important approach for the XCell generators is it has to produce a DRC clean layout as it is important that the generated XCells are reliable and do not need manual

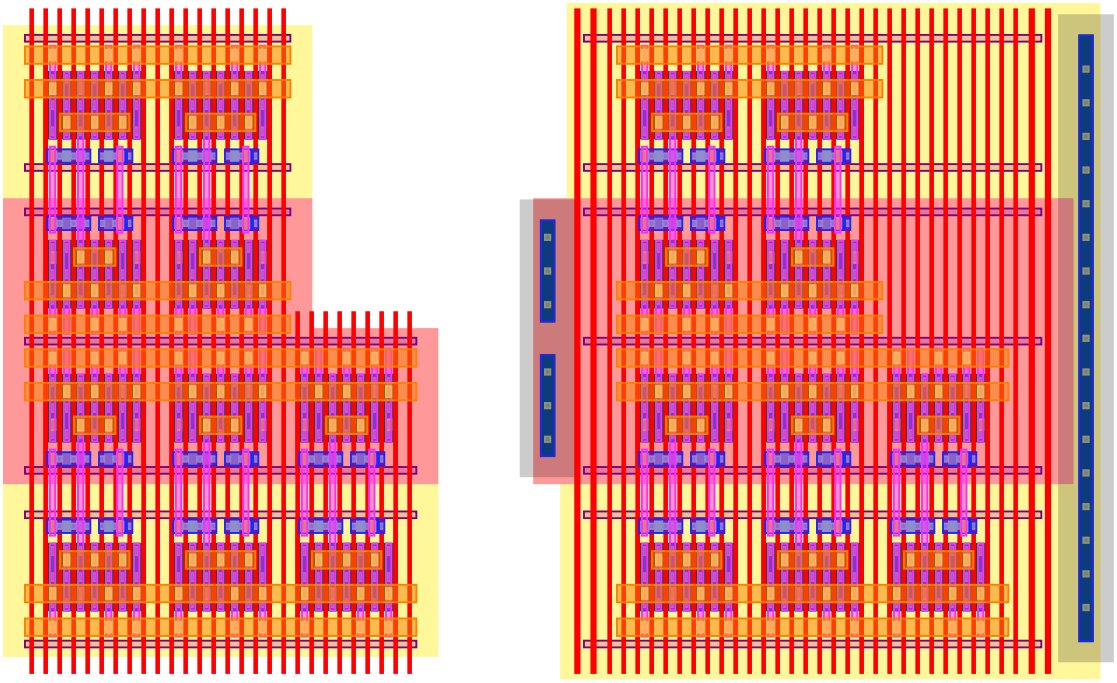


Figure 6.21: CMOSPost constraint, left: target, right: generated

modification after generation. To achieve this a test deck is introduced including multiple different constraint setups to test different aspects of the generator.

In the XCell framework, it is possible to execute the generators in a coverage mode using coverage.py [69] instead of the normal python interpreter resulting in a coverage measurement of the executed code.
With help of the resulting coverage data, a first test deck is build to get a 100% code coverage to ensure that every branch in the code was taken and executed. This approach does not find all bugs but every line of the code is at least executed once. Afterward, the test deck is extended with corner or special cases for the generator which come to mind i.e. single finger members, large length devices.
This results in a good initially tested generator code. Overtime when bugs are found by the layout designers the bug case is reduced to a small example and added to the test deck. The test deck is run with each release of the XCell framework or the tested generator library.

The XCell framework and the different generator libraries are deployed as python-pip packages. The releases get a version and the user can work with the pip package manager

to install packages or select specific versions. When a tape out takes place the current packages can be frozen via pip. This is important as with this information the state of the tape out can be reliably reproduced.

7 Results

7.1 Skillbridge and Schematic Automation

In this work, the skillbridge was developed as an interface to Virtuoso and used as a foundation for further vendor-independent schematic and layout automation.

The skillbridge was released as an open-source project and several companies and universities started using skillbridge as a foundation to replace their skill code or to evaluate python as the interfacing language to the ASIC design system. This can be seen from the interactions within the skillbridge repository on the one hand issue were raised for potential enhancements regarding different use cases from automating the simulation interface to rewriting to scripts for automatic pin generation and placement in the layout. These use cases and the resulting issues for the skillbridge are immensely helpful as they allow to further mature the library.

This work implemented hierarchical structure generation which writes the structural schematics was tested in large-scale designs without any issues.
The parametrized schematic leaf generation was not used in larger projects yet. For the leaf generation, some examples were introduced and dimensioned. One problem for the parametrized leaf generation is that the circuit designer has to implement the dimensioning scripts in python, which is normally done manually on paper or in another document. Writing these scripts seems like additional work which makes it difficult to introduce them as a normal task. On the other hands, these scripts automatically can be used as executable documentation.

7.2 Layout Automation

7.2.1 Elementary Cells

The elementary cells were used in larger projects. From one of these implementations, the usage is analyzed in detail. For this, a little statistic about the drawn shapes in the layout was created with the skillbridge.

For this 35 leaf cells where PCell elementary cells were used are analyzed. In these cells, the shapes of the leaf cell are read out as shapes set by the user. Shapes are organized on the one hand by the connectivity. VDD and VSS nets are counted as power shapes, shapes without connectivity as dummy, and all other shapes as routing shapes. The second category is depending on the shapes layer M1 and M2 are categorized as local layers as they are both double patterned. The C1-C6 metal layers are categorized as middle metal layers (mid) and IA-IB as top metal layers (top). Afterward, each elementary cell in the leaf cell view is read out and its master shapes are added to the generated shape category. Additionally, the shapes of the BASIC-FET instances inside the elementary PCell are added to the generated shapes. Shapes within the PDK devices are not counted.

In figure 7.1 the results can be seen. The absolute accumulated shapes with their layer category and generation/ usage category are shown. It is observed that especially for the local layer category the amount of generated shapes compared to the user shapes is about 80%, for the mid category 40% none for the top category and around 10% for the other category.

While from this analysis an absolute time saving during the layout process cannot be derived it can be said that time was saved and this saving must be large. For one this is the case since the local route category has the most DRC rules. In manual layouts, these shapes often lead to many reiterations which are very time-consuming.
In these results, it can further be seen that only a tiny fraction of the missing user shapes are shapes for routing. Most shapes are part of dummy metal or the power grid.
The relation between power, dummy, and routing shapes stay the same in the higher levels of the design hierarchy.
In figure 7.2 an example layout builds up from elementary cells can be seen. Here the full layout of a phase interpolator is shown, consisting of a mixer, output buffer, I and Q DAC, cross-domain buffer, and offset bits.

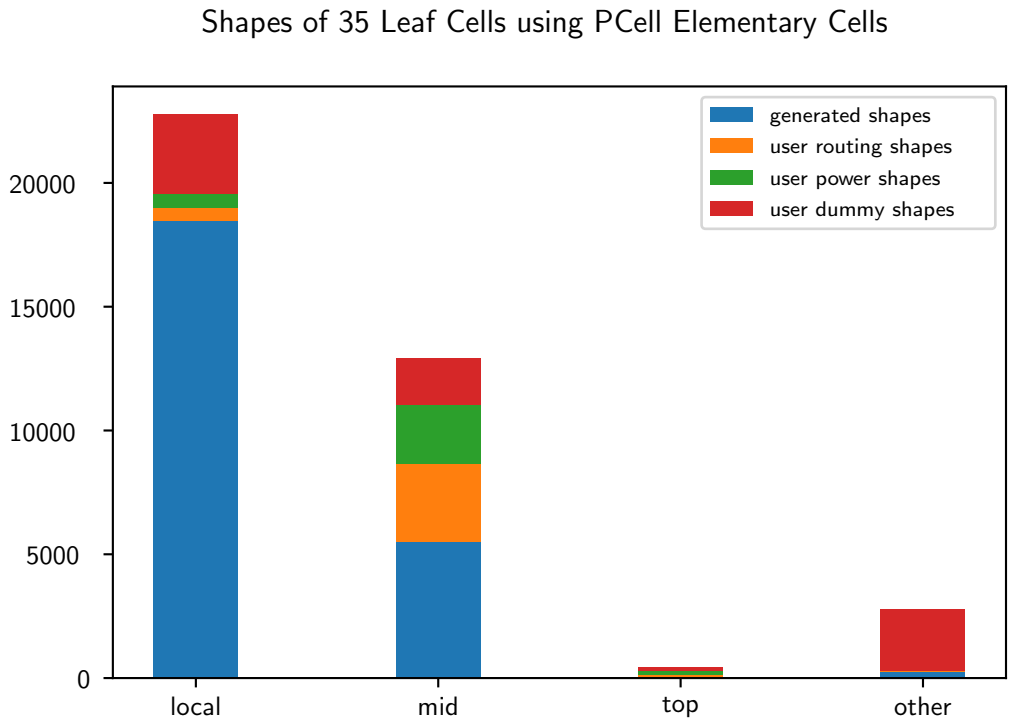


Figure 7.1: Shape statistics over different leaf cells binned by layer group and category

In almost all leaf cells elementary cells are used. The only exception is the XDomain buffers, which are CMOS-based. In the design, it can be seen that the mixer consists of multiple transistor pair elementary cells, the output buffer out of one transistor pair, and a transistor array. Both current source bits consist of a transistor pair and a transistor array, both with 2 rows but with a different number of fingers.

Only the leaf cells with PCells were analyzed. The other cells can be sorted into the following categories. The design consists of three main blocks and one of the main blocks was already designed several months in advance when the elementary cells were not available yet resulting in multiple leaf cells which do not use elementary cells but are a perfect fit. Also in the design, a trend can be seen to more full custom CMOS logic, and thus several leaf cells are CMOS based. The last category is very specific analog leaf cells, fill cells, decap, or passives.

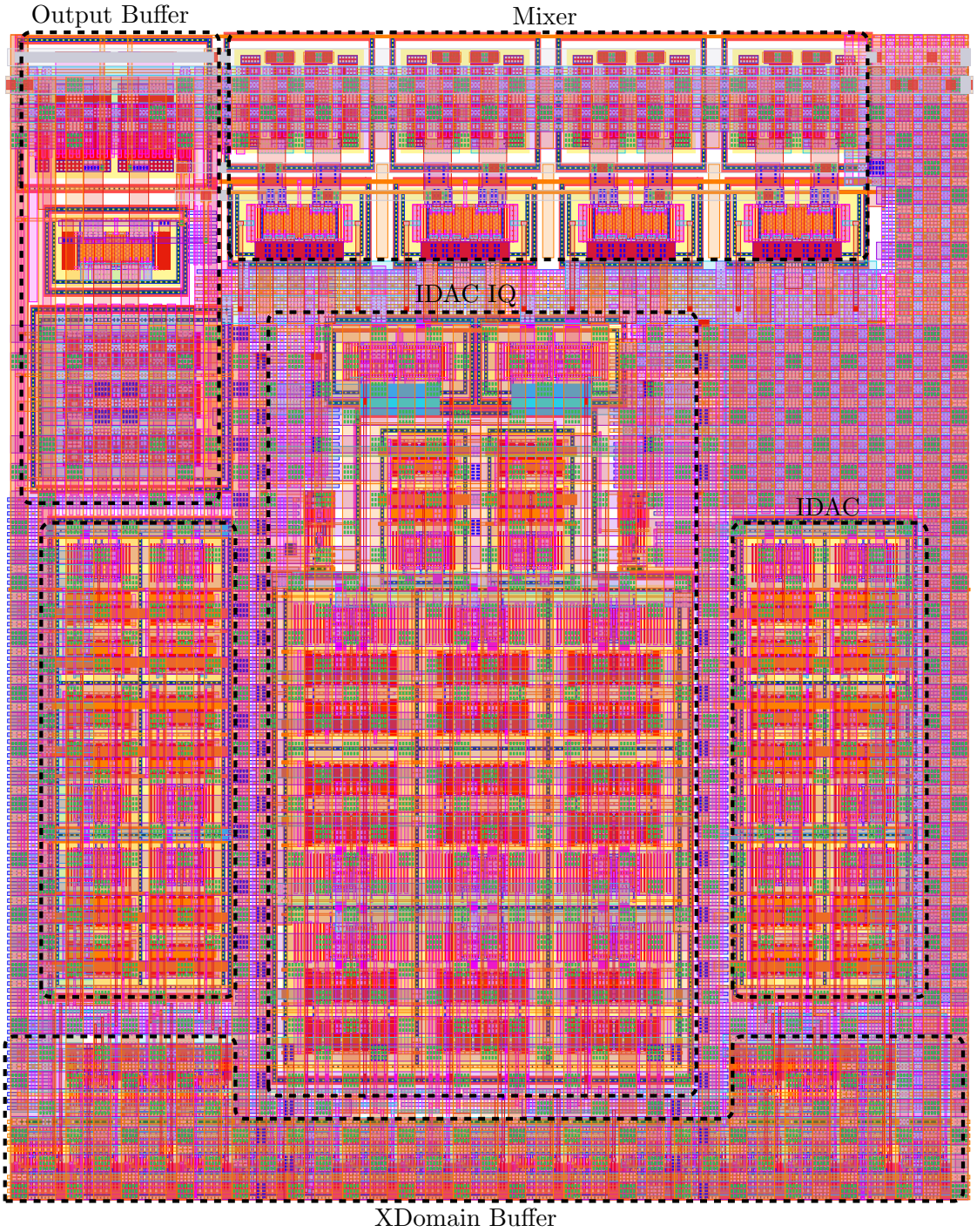


Figure 7.2: Phase Interpolator using PCell elementary cells

7.2.2 XCells

With the XCells the elementary cells are simpler to use and some drawbacks regarding DFM especially density issues based on the guard rings were eliminated. Furthermore, the approach to separate guardrings from the elementary cells allows to simply create well taps or rings which will be abutable which results in better integration. Furthermore, the constraint-based approach proved to be more flexible for both user and project author in terms of member selection and usage inside the constraint generator.

Furthermore as observed while analyzing the shapes of leaf cells where PCells were not used one of the main groups where CMOS based leaf cells. With the XCell framework, CMOS generators are available too. As soon as the XCell generators were available several leaf cells were implemented with them. Furthermore, also some CMOS cells were implemented. In figure 6.18 a CMOS divider is shown which divisor can be switched between 2 and 3. The Divider consists of several latches and NAND leaf cells.
On the left, in the figure, the originally implemented divider is shown. On the right, the divider is implemented with the XCell CMOS generator. The NAND cell consists of a single CMOS constraint, while the latch is build up from 2 CMOS constraints where all devices are constraint members except the minimal cross-coupled inverters. While in the NAND nearly every shape except the pin shapes are generated in the LATCH the cross-coupled inverters are inserted between two CMOS constraints by hand and the interconnect in between is done manually.
In the top-level of the divider, a CMOS post constraint is defined resulting in a DRC clean layout. The interconnect in the top-level is done manually.

The resulting divider layout is smaller than the initial layout block. There are several reasons for this. For one the old divider was implemented early when the technology was new and expert knowledge did not accumulate yet. The layout could be done more optimal by using poly cut layers correctly. The second main point in area reduction is that the left design implements the well-contacts in each cell resulting in simply abutable DRC clean designs but also in a larger area. In the XCell approach, all devices were abutted without an individual well contact, and then afterward the post constraint generates the DRC clean casing and wells. This does not only save multiple well-contacts but also the stop dummies since each cell does not have to be DRC clean. Such an approach is also possible in manual layout but in terms of design time not feasible since each casing has to be implemented manually. In this case, the XCell generator approach combines the best of both worlds.

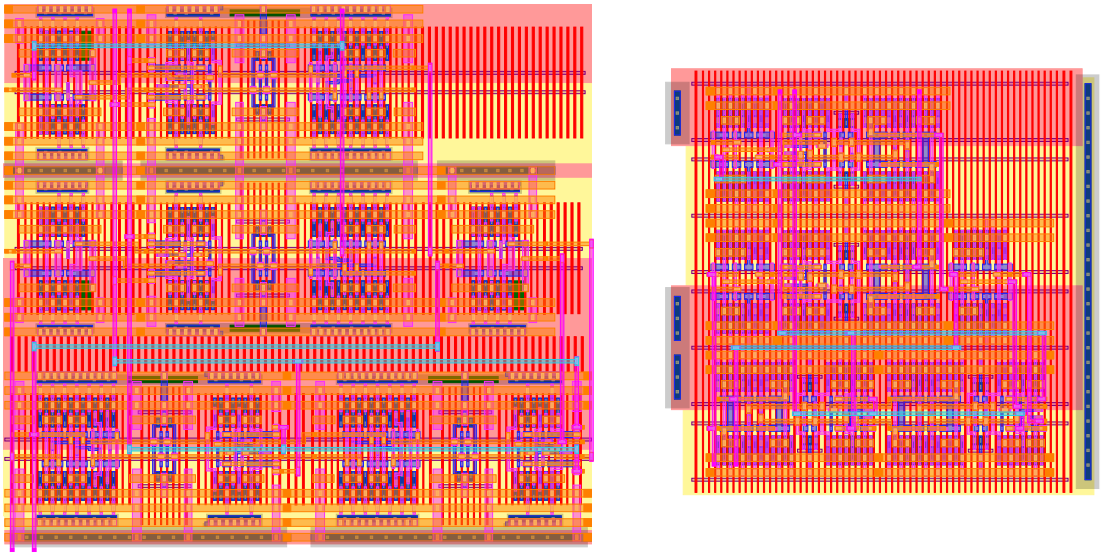


Figure 7.3: Divide by 2/3 Divider, left: original layout, right: XCell CMOS constraint based layout

Another important aspect for XCell generators is the formalization of expert knowledge. As in the example, seen knowledge can be incorporated into the generator. This has then the advantage that this knowledge is applied to every layout-block where the constraint is used. Furthermore, the layout generators implemented with the developed XCell framework are technology agnostic and were tested with a 22 nm FDSOI and 28nm bulk technology and thus further increase automation.

Additional to the use of the XCell generators the XCell framework was also used to implement additional generators.

Two of these generators are the custom fill constraint and the power grid constraint. Both generators determine obstacles in the layout and add accordingly fill or create a power grid. For this, the power grid generator reads out the connectivity of the existing shapes and connects to the highest VDD, VSS shapes of the in the cellview existing instances.

Also, an inductor generator constraint was developed which similar to the snake resistor example can be applied to an LVS resistor to create an inductor from it.

7.2.3 Comparison to PCell Designer and BAG

When comparing the XCell framework to PCells its advantages become clear as it is inherently technology agnostic. Furthermore, the constraint-based approach allows incorporating multiple factors which cannot simply be used inside a PCell for example the circuit connectivity or existing obstacles in the layout. Furthermore, the XCell framework solves all the issues and lessons learned from the PCell designer as the side effects and code reuse. Due to this the Xcell Framework and its created generators completely replaced the PCell Designer and the elementary cells implemented with it.

In comparison to the BAG and BAG2 mentioned in the State of the Art chapter, the XCell framework and generators are a different approach. For one with the XCell framework the goal is to constraint existing devices for layout generation this can be done hierarchically and with inter depending constraints resulting in a good constraint design for layout generation.
Indifference the BAG approach is to implement the complete generators which itself is a very complex task but is further complicated since each PDK device must also be implemented and cannot be used as given by the vendor. This especially when different technologies are used is a big disadvantage since for each many devices have to be implemented.
While the BAG framework implements complete generators directly in the XCell framework complete generators would be implemented as a hierarchical layout constraint description thus being user-friendly in constraining.
Furthermore, XCell offers well-defined interfaces and abstraction to create technology-agnostic generators while BAG only offers little abstraction by itself.

The XCell framework and the implemented generators are simple to use, technology agnostic and result in performant layouts which fulfill DFM requirements.

8 Conclusion

With the skillbridge a python $< - >$ Skill interface was introduced which was widely adopted and has become an active open-source project used by academia and industry. Skillbrigde is a seamless language mapping that dynamically maps methods and types. This means even when skill subsystems are added to virtuoso, skillbridge can access them with its dynamic approach without any changes necessary.
With the skillbridge as foundation schematic automation and layout automation tools were implemented.

For schematic automation, a design flow was implemented which allows creating sized circuits from the system level parameters. This automated flow guarantees consistency between system-level and implementation. Furthermore, the technology-agnostic system-level is separated from the implementation. Classes and an internal design representation are added to enable documenting and making expert knowledge executable. The implemented sizing scripts are technology-agnostic since dependent technology values are accessed via an interface to technology-dependent simulated result tables.

With the XCell framework, a constraint-based layout generator approach was implemented which allows well-defined interfaces to be technology and tool agnostic. Furthermore, the XCell framework allows implementing expert knowledge in terms of layout.

The with the XCell framework implemented elementary cells and other generators offer good automation and generate a significant part of the overall layout. The implemented generators are parametrizable and very versatile increasing design efficiency by reducing the individual layout blocks to a single constraint.
The constraint-based approach further proved to be simple to use and very flexible as constraints can interactively be marked on the constraint members and parameters can be set. Additionally, XCell generators allow an explorative layout approach where different parameter sets for the generators can simple be tested with the comfortable generate, clean, and regenerate mechanisms.

Another advantage of the hierarchical constraint-based approach is that some workflows in the layout become possible as dense arrays may be finished with a post cell as seen in the CMOS example. This leads to overall denser layouts.

8.1 Future Work

8.1.1 Schematic Generation

For the schematic generation, some glue code to ensure smooth usage has to be implemented. This also includes embedding triggers and GUI elements for different generation modes and template generation into the ASIC design environment.

Furthermore, the testbench setup should also be automated further. With this, there might be a chance to automate iterations for dimensioning or use the initial circuit values to set up a corner optimization without having to modify the sizing scripts to incorporating technology tables from multiple corners and thus simplifying the sizing scripts for the designer.

8.1.2 Layout Generation

For layout generation, the XCell framework allows for multiple generators further automating the layout. One aspect is that with the hierarchical constraints complete generators can be implemented. For this, especially constraints for placement and routing on cell level and system level have to be implemented. Furthermore as in the result chapter was observed Decap cells could be implemented as a constraint and thus generating as much Decap as possible in a cell.

Another thing to do is to implement additional smaller nodes and see if a FinFet technology can be mapped as a specific technology and if not implement the missing features. In a first estimate, it should be possible. Only the local routing group gets extended by M0 which is an additional layer often used in small FinFet nodes.

The formalizing of expert knowledge into constraint-based generators works excellent but especially with the increase in DRC rules finding optimal solutions is increasingly difficult. A good approach is to implement mixed approaches including expert-based descriptions and optimizer-based approaches for solving sub-problems.
Implementing solvers accordingly and making them available to the user in a simple

manner would bring benefits especially in tasks like placing or routing.
Furthermore, it was observed that one remaining time-consuming task whether to implement them within an XCell generator or manually are little geometric problems caused by the complex DRC rules. One example is the Via Problem from the first chapter.

These small geometric problems have to be solved within the constraint of the complex DRC rules. These problems are constraint solving problems. A good solution could be implementing a 2D geometric constraint solver where the constraints are the DRC rules and the optimization goal. For both of these, an interface has to be offered to simply define these goals and rules as a user.

List of Figures

Bibliography

[1] en.wikipedia.org, "Transistor count," 2019. https://en.wikipedia.org/wiki/Transistor_count.

[2] en.wikichip.org, "Zen - microachitectures - amd," 2021. https://en.wikichip.org/wiki/amd/microarchitectures/zen#Core_2.

[3] S. Naffziger, K. Lepak, M. Paraschou, and M. Subramony, "Amd chiplet architecture for high-performance server and desktop products," in *Presentation 2020 IEEE International Solid- State Circuits Conference - (ISSCC)*, pp. 44–45, 2020.

[4] F. Silveira, D. Flandre, and P. Jespers, "A gm/id based methodology for the design of cmos analog circuits and its application to the synthesis of a silicon-on-insulator micropower ota," *Solid-State Circuits, IEEE Journal of*, vol. 31, pp. 1314 – 1319, 10 1996.

[5] M. J. M. Pelgrom and A. C. Duinmaijer, "Matching properties of mos transistors," *ESSCIRC '88: Fourteenth European Solid-State Circuits Conference*, pp. 327–330, 1988.

[6] M. White, "Are you (really) ready for your next node," 2017. https://blogs.sw.siemens.com/calibre/2017/01/11/are-you-really-ready-for-your-next-node/.

[7] B. Hoefflinger, "Itrs: The international technology roadmap for semiconductors," in *Chips 2020*, pp. 161–174, Springer, 2011.

[8] W. Savage, "The unintended consequences of massive ip reuse," 2020. https://www.chipestimate.com/The-Unintended-Consequences-of-Massive-IP-Reuse/Silvaco/Technical-Article/2016/04/19.

[9] R. Collett and D. Pyle, "What happens when chip-design complexity outpaces development productivity?," 2013.

[10] I. APT Technologies, D. C. Corporation, I. Corporation, I. Corporation, M. Corporation, and S. Technology, "Serial ata: High speed serialized at attachment," 2001.

[11] S.-I. B. Members, D. C. Corporation, H. P. Corporation, HGST, I. Corporation, M. Semiconductor, P.-S. Inc., S. Corporation, S. Technology, and W. D. Corporation, "Serial ata international organization serial ata revision 3.2," 2013.

[12] M. Wood, D. Boppana, and I. Land, "Jesd204a for wireless base station and radar systems," 2010.

[13] I. Poole, "Usb standards: Usb 1, usb 2, usb 3, usb 4 - capabilities and comparisons," 2020. https://www.electronics-notes.com/articles/connectivity/usb-universal-serial-bus/standards.php.

[14] A. Dhamba and A. V. Kulkarni, "Design considerations for high bandwidth memory controller," 2020. https://www.design-reuse.com/articles/41186/design-considerations-for-high-bandwidth-memory-controller.html.

[15] Rambus, "The ultimate guide to hbm2e implementation and selection," 2020. https://www.rambus.com/blogs/hbm2e/.

[16] I. Micron Technology, "Hybrid memory cube - hmc gen2," 2018.

[17] J. J. Maki, "Ethernet adoption: Serdes rates and form factors," 2016.

[18] en.wikipedia.org, "Pci express," 2020. https://en.wikipedia.org/wiki/PCI_Express.

[19] computerhistory.org, "Rubylith operators," 2020. https://www.computerhistory.org/revolution/artifact/287/1614.

[20] cadence, "Virtuoso layout suite," 2020. https://www.cadence.com/en_US/home/tools/custom-ic-analog-rf-design/layout-design/virtuoso-layout-suite.html.

[21] A. Olofsson, "Silicon compilers-version 2.0," 2018.

[22] J. Chen, M. Henrie, M. Mar, and M. Nizic, *Mixed-Signal Methodology Guide*. Cadence Design System, Incorporated, 2012.

[23] M. Mueller, *Digital Centric Multi-Gigabit SerDes Design and Verification*. PhD thesis, 01 2018.

[24] M. Schweikardt, Y. Uhlmann, F. Leber, J. Scheible, and H. Habal, "A generic procedural generator for sizing of analog integrated circuits," in *2019 15th Conference on Ph.D Research in Microelectronics and Electronics (PRIME)*, pp. 17–20, 2019.

[25] scientific analog, "Scientific analog webpage," 2021. https://www.scianalog.com/.

[26] J. Crossley, A. Puggelli, H. . Le, B. Yang, R. Nancollas, K. Jung, L. Kong, N. Narevsky, Y. Lu, N. Sutardja, E. J. An, A. L. Sangiovanni-Vincentelli, and E. Alon, "Bag: A designer-oriented integrated framework for the development of ams circuit generators," in *2013 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pp. 74–81, 2013.

[27] ucb art, "Bag_framework," 2020. https://github.com/ucb-art/BAG_framework.

[28] E. Chang, J. Han, W. Bae, Z. Wang, N. Narevsky, B. NikoliC, and E. Alon, "Bag2: A process-portable framework for generator-based ams circuit design," in *2018 IEEE Custom Integrated Circuits Conference (CICC)*, pp. 1–8, 2018.

[29] B. Prautsch, U. Eichler, S. Rao, B. Zeugmann, A. Puppala, T. Reich, and J. Lienig, "Iip framework: A tool for reuse-centric analog circuit design," in *2016 13th International Conference on Synthesis, Modeling, Analysis and Simulation Methods and Applications to Circuit Design (SMACD)*, pp. 1–4, 2016.

[30] cadence, "Introduction to skill pcell programming," 2020.

[31] O. Broschart, "Automated design and implementation of a ddr4/lpddr4 receiver," 2019.

[32] "jupyter webpage." https://jupyter.org/. Accessed: 2020-11-29.

[33] T. Ajayi, D. Blaauw, T. Chan, C. Cheng, V. A. Chhabria, D. K. Choo, M. Coltella, R. Dreslinski, M. Fogaça, S. Hashemi, A. Ibrahim, A. B. Kahng, M. Kim, J. Li, Z. Liang, U. Mallappa, P. Penzes, G. Pradipta, S. Reda, A. Rovinski, K. Samadi, S. Sapatnekar, L. Saul, C. Sechen, V. Srinivas, W. Swartz, D. Sylvester, D. Urquhart, L. Wang, M. Woo, and B. Xu, "Openroad: Toward a self-driving, open-source digital layout implementation tool chain," 2019.

[34] "magic vlsi layout tool webpage." http://opencircuitdesign.com/magic/. Accessed: 2020-11-29.

[35] S. Maurya, *A Structured Design Methodology for High Performance VLSI Arrays*. Arizona State University, 2012.

[36] S. A. Vitale, P. W. Wyatt, N. Checka, J. Kedzierski, and C. L. Keast, "Fdsoi process technology for subthreshold-operation ultralow-power electronics," *Proceedings of the IEEE*, vol. 98, no. 2, pp. 333–342, 2010.

[37] H. Cedric and S. Selberherr, "Electromigration in submicron interconnect features of integrated circuits," *Materials Science and Engineering:R:Reports*, pp. 53–86, 05 2013.

[38] J. Lienig and M. Thiele, *Fundamentals of Electromigration-Aware Integrated Circuit Design*. Springer Publishing Company, Incorperated, 1st ed., 2018.

[39] J. R. Black, "Electromigration failure modes in aluminum metallization for semiconductor devices," *Proceedings of the IEEE*, vol. 57, no. 9, pp. 1587–1594, 1969.

[40] W. Li and C. M. Tan, "Black's equation for today's ulsi interconnect electromigration reliability — a revisit," in *2011 IEEE International Conference of Electron Devices and Solid-State Circuits*, pp. 1–2, 2011.

[41] H. H. Hoang, "Effects of annealing temperature on electromigration performance of multilayer metallization systems," in *26th Annual Proceedings Reliability Physics Symposium 1988*, pp. 173–178, 1988.

[42] S. Vaidya, T. Sheng, and A. Sinha, "Linewidth dependence of electromigration in evaporated al-0.5% cu," *Applied Physics Letters*, vol. 36, no. 6, pp. 464–466, 1980.

[43] C. S. Hu, R. Rosenberg, and K. Y. Lee, "Electromigration path in cu thin-film lines," 1999.

[44] I. A. Blech, "Electromigration in thin aluminum films on titanium nitride," *Journal of Applied Physics*, vol. 47, no. 4, pp. 1203–1208, 1976.

[45] cadence, "Verilogin quick reference to basics and most referred solutions," 2020.

[46] rbzentrum, "Spam," 2020. https://github.com/rbzentrum/SPAM.

[47] "Tiobe index for january 2021." https://www.tiobe.com/tiobe-index/. Accessed: 2021-01-21.

[48] "Githut 2.0." https://madnight.github.io/githut/#/pull_requests/2020/4. Accessed: 2021-01-21.

[49] "Most popular technologies stackoverflow survey 2020." https://insights.stackoverflow.com/survey/2020#most-popular-technologies. Accessed: 2021-01-21.

[50] unihd cag, "skillbridge," 2020. https://github.com/unihd-cag/skillbridge.

[51] "pycharm webpage." https://www.jetbrains.com/de-de/pycharm/. Accessed: 2021-03-05.

[52] "theopenroadproject webpage." https://theopenroadproject.org/. Accessed: 2021-03-05.

[53] cadence, "Spectre ams designer and xcelium simulator mixed-signal user guide 20.09," 2020.

[54] "Ieee standard for systemverilog–unified hardware design, specification, and verification language," *IEEE Std 1800-2012 (Revision of IEEE Std 1800-2009)*, pp. 1–1315, 2013.

[55] cadence, "Genus synthesis solution," 2021. https://www.cadence.com/ko_KR/home/tools/digital-design-and-signoff/synthesis/genus-synthesis-solution.html.

[56] J. Nagel, "Parsing and handling directives in a system-verilog intellij plugin," 2017.

[57] "yosys webpage." http://www.clifford.at/yosys/. Accessed: 2021-03-05.

[58] M. Orshansky, S. Nassif, and D. Boning, "Design for manufacturability and statistical design: A constructive approach," *Design for Manufacturability and Statistical Design: A Constructive Approach*, pp. 1–316, 01 2008.

[59] A. R. Subramaniam, R. Singhal, Chi-Chao Wang, and Yu Cao, "Design rule optimization of regular layout for leakage reduction in nanoscale design," in *2008 Asia and South Pacific Design Automation Conference*, pp. 474–479, March 2008.

[60] T. Jhaveri, L. Pileggi, V. Rovner, and A. J. Strojwas, "Maximization of layout printability/manufacturability by extreme layout regularity," in *Design and Process Integration for Microelectronic Manufacturing IV* (A. K. K. Wong and V. K. Singh, eds.), vol. 6156, pp. 67 – 81, International Society for Optics and Photonics, SPIE, 2006.

[61] W. Maly, Y. Lin, and M. Marek-Sadowska, "Opc-free and minimally irregular ic design style," in *2007 44th ACM/IEEE Design Automation Conference*, pp. 954–957, June 2007.

[62] M. Pons Solé, "Layout regularity for design and manufacturability," 2012.

[63] "Overview litho physical analyzer." https://www.cadence.com/en_US/home/tools/digital-design-and-signoff/silicon-signoff/litho-physical-analyzer.html. Accessed: 2021-02-23.

[64] N. Bergmann, "Generalised cmos - a technology independent cmos ic design style," in *22nd ACM/IEEE Design Automation Conference*, pp. 273–278, June 1985.

[65] unihd cag, "simple-geometry," 2020. https://github.com/unihd-cag/simple-geometry.

[66] unihd cag, "rodlayout," 2020. https://github.com/unihd-cag/rodlayout.

[67] T. Markus and N. Buwen, "Partial layout generation with python," 2020. Internal design document.

[68] cadence, "Virtuoso unified custom constraints user guide icadvm20.1," 2021.

[69] nedbat, "coveragepy," 2021. https://github.com/nedbat/coveragepy.