Automated Creation of Reusable Generators for Analog IC Design with the Intelligent IP Method*

Uwe Eichler, Benjamin Prautsch, Torsten Reich, Fraunhofer IIS/EAS, Dresden, Germany
{forename}.{surname}@eas.iis.fraunhofer.de

***Abstract*—Procedural generators are often proposed for analog IC design automation. They promise to encapsulate designer knowledge and intellectual property (IP) data in a deterministic and reusable way. While recent developments claim to have proven this, one question remains: How to create generators efficiently and integrate them in an automated design flow? A major challenge for generators is the trade-off between initial implementation effort, reusability, and acceptance. This raises further questions on the role of the generator supplier: Who should spend the effort implementing and maintaining generator IP? Which interfaces and standards can be used to implement and integrate them into common design environments? In order to address these challenges, we propose a combination of pre-defined generators for basic building blocks at lower hierarchy levels with automatic creation of generators using place and route templates for more complex circuits. The paper demonstrates the successful application of this flow to an OTA design and discusses the required implementation efforts, quality of the generated results and potential future developments.**

Keywords—analog; layout; generators; templates; design automation; reuse; soft IP

I. Introduction

Analog and custom IC design is an increasing bottleneck in SoC (System-on-Chip) development even though its share in terms of number of transistor devices is rather low [1]. While circuit and layout synthesis based on hardware description languages and standard cell libraries is state of the art in digital IC design for decades, methods to automate analog IC design and full-custom layout are not well established.

A main differentiator of analog design automation approaches is the way they translate input requirements to a layout. There are two basic methodological directions: generators and synthesis. A generator is a programmatic description of a circuit that, upon execution, creates circuit design data such as schematic and layout in a deterministic way. Parameters control rather low-level properties such as topology, sizing, placement, and routing. Generators typically create individual circuit classes and are thus organized in a generator library [3][5][9]. Synthesis tools, in contrast, have a broader focus. They search for a placement and routing solution for a wide range of circuit classes based on a set of constraints usually derived from the circuit netlist. Often, synthesis algorithms use optimization also considering electrical performance parameters of the circuit [8]. While generators reduce the complexity of the solution space by offering a well-selected set of variants to the designer, synthesis handles this complexity by efficiently searching in the solution space for feasible variants.

As complexity is usually of different nature across different circuit hierarchy levels [2], both automation approaches are often combined when larger circuits are to be handled [1][10]. Basic analog building blocks can typically be constructed from unit-size transistors or passive devices and often have matching constraints which makes their array-style layouts well scalable and easy to implement in a generator. In the next higher hierarchy level (e.g. analog functional blocks), these array blocks then have to be arranged and routed in an area-efficient and parasitic-aware way. This also retains some degree of flexibility that is again required for the next higher level. Due to the large number of possible layout solutions at these higher hierarchies, synthesis approaches tend to be useful here. Simultaneously, generators can provide the underlying layout engines that are able to create many layout variants already fulfilling basic constraints such as DRC and LVS. These hierarchical generators can then be controlled either by designers directly or by an optimizer.

* This work was enabled by the project InnoStar (grant no. 16ME0462), funded by the German Federal Ministry of Education and Research (BMBF).

This paper proposes a hierarchical generator approach based on the Intelligent IP Framework (IIP) [5] that uses a library of pre-defined base-level generators together with a tool for automatic creation of generators for higher hierarchy levels. Comparable to schematic-driven layout flows (SDL), input is a schematic representation of the circuit. The difference is that the newly created generators may then be used solely to generate further variants of the design including all schematics and layouts.

II. The Intelligent IP Approach

Intelligent IP (IIP) is the name of a software that provides the infrastructure to implement and execute circuit generators within an custom IC design environment such as Cadence Virtuoso. In IIP, a generator is a program that creates design data (usually so-called cell views) of a circuit building block in the design database of the design environment used. IIP generators are focused on a rather *deterministic* but also parameterizable as well as PDK-agnostic description and generation of consistent design data (rather than focused on *searching* solutions in the design space according to target performances). Although it would be possible to integrate sizing or optimization algorithms into the generator framework or a specific generator, the current approach intends to provide interfaces to existing tools in order to enable functionality like layout-aware sizing with generators in the loop. Generic interfaces to process technology data defined by a PDK (process design kit) and to the design environment allow a well portable generator implementation. The generator programs are written in Python based on the IIP API (application programming interface) that also supports complex parameter dependencies and hierarchy (see Figure 3). Together with a built-in library of basic generators, this allows writing comprehensive hierarchical generators that can create the full hierarchy of a circuit’s design data only based on this program [5][6][7].

The built-in basic generators cover the two lowest hierarchy levels. At device level, there are wrapper generators for primitive devices of the PDK (MOS transistors, capacitors, resistors, etc.) that map the technology-specific PCell instances and their parameters to a generic subset of device configurations used in IIP. One level above, there are generators for very common structures such as capacitor and resistor arrays, matching MOS arrays for current mirrors and many other topologies, and a differential pair (see Figure 1).

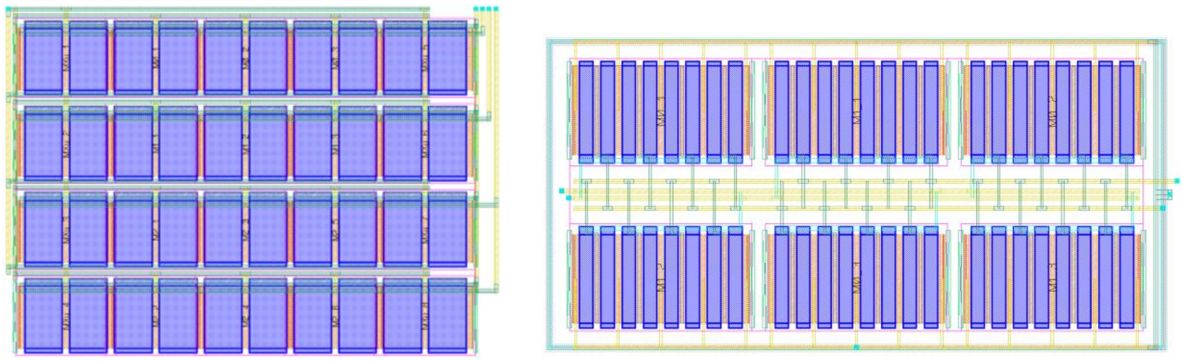


Figure 1: Examples of IIP base-level generators MosArray (left, supports many options for topology, pattern, and routing) and DiffPair (right, symmetric place pattern and routing).

III. Generator-based Design methodology

For the automation of design hierarchy levels above basic structures, there are the following generator-based usage scenarios with different trade-offs between implementation effort and reusability:

- Using generators up to base-level only (left side of Figure 2) already increases sizing and layout productivity but limits the reuse because place and route of the levels above have to be done manually.
- A custom, circuit-specific generator implementation with low-level description of instances, wires, vias, and pins would either be less flexible especially regarding layout arrangements or would require large implementation efforts with less reusable code to produce customizable and area-efficient layouts.

- Abstract layout templates [7] that define relative positions of sub-block instances and interconnects in a regular way might be reused even across hierarchies and thus can significantly reduce the implementation effort of a generator's layout part. Their regularity and universality makes them well suited for automatic creation of generator code that supports a wide range of technologies. Proving the feasibility of such generic layout styles for circuit performances is one aspect of this work.

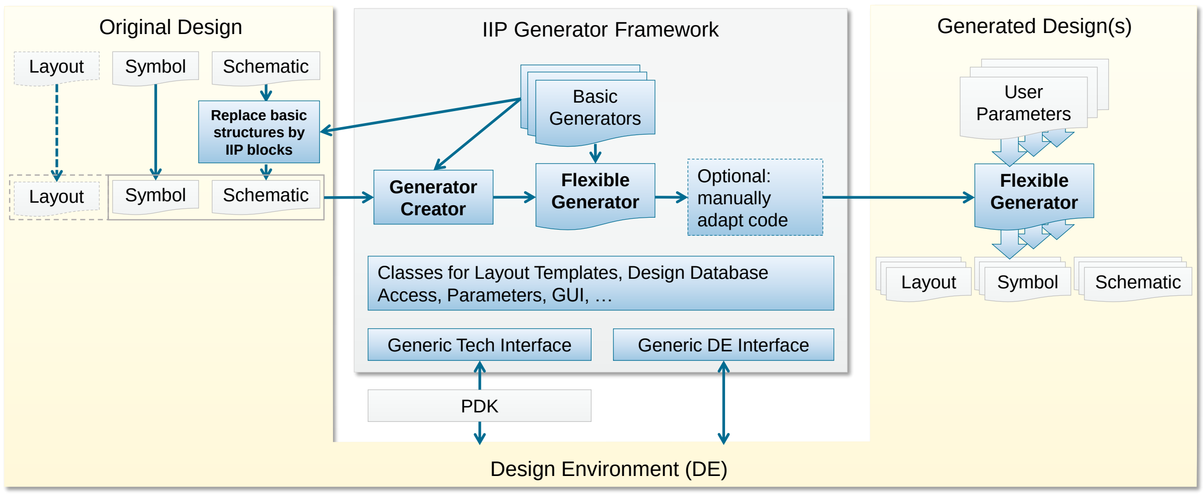


Figure 2: Overview of the generator creation flow when using the Intelligent IP approach.

Before this work, the IIP Creator tool (Figure 2, Figure 5 right) was already able to generate code of a new IIP generator based on an existing schematic and symbol of a circuit block, also for several cells in a design hierarchy in one step. The new generator got parameters for all the sub-block or device instances found in the schematic and contained code for re-generating the symbol, the schematic, and a simplified layout view in a technology-independent and parameterizable way. In the layout part, it created a simple side-by-side arrangement of all the sub-block layouts without any routing. The original target applications of the IIP Creator were: creating code templates for generator development, and porting design data between technologies or design environments. While both still work in many cases, the main drawback was the limited layout support. The generator developer had to add meaningful algorithms for place and route or, when the generator was used unchanged, the top-level of the generated layout had to be re-drawn by a layout engineer.

In order to overcome these limitations, we combined the layout template approach with the IIP Creator such that the initially generated layout code is already able to produce appropriate layouts. In a previous work, we already investigated chessboard-like templates (which we called MESH [6]) and applied them to array layouts of switches [6] and capacitances [7]. In both cases, a larger number of unit-size elements were placed which made it easier to find an area-efficient placement and routing solution. Here, we wanted to support also layouts with fewer but differently sized sub-blocks such as operational amplifiers. In a MESH floorplan, which is sliceable along all edges in both directions, the largest block would define the height of a row and the width of a column resulting in unused area in all other cells with smaller blocks. This could be addressed by either hierarchical templates that can sub-divide cells of a floorplan into smaller cells or, what we did here, by another, less restrictive template. We chose a so-called "street" floorplan similar to [4] that places all blocks in up to two rows along a central routing channel (see Figure 7). It is only sliceable along one axis such that sub-blocks of arbitrary width might be arranged together in one row and only their heights could be aligned when optimizing for area. The routing was simplified to a single channel containing parallel wires for all required signals and straight connections to the sub-block pins. To avoid conflicts when accessing the channel from both rows within short horizontal distance, up to three different layers are used, and the pins of the sub-blocks are configured accordingly (see Figure 8).

As a result, the user may now read-in a schematic design into the IIP Creator and will receive a generator that is able to not only re-generate the schematic and sub-block layouts for different parameters or technologies but also a top-level layout that can be configured regarding placement of sub-blocks within the street template and several routing properties. This layout is then correct in terms of DRC and LVS and can be used in subsequent design steps.

When looking at a code example (see Figure 3), it becomes obvious that the templates are also a great help for generator developers. Behind the few lines for configuration and executing the template within the generator's layout method, there are currently several thousand lines of code for placement and routing the developer would otherwise have to replace by individual code.

```
# this is iiplib.std.Ota1
import iip…    # API
import iiplib…    # sub-generators
class Generator(iip.gen.HierBlock):
    def param_spec(self):  # define parameters, their constraints, and init dependent class members
        # add constrained parameter(s)
        self.params.add("nRows", 2, "number of template rows", RangeConstraint(1, 2))
        …
        # add sub-generator(s), here for DiffPair with initial parameter values
        self.generators.add("dp", iiplib.base.DiffPair, Params(w="1u", l="300n"))
        …
        # add proxy parameter(s), here from sub-generator dp (hierarchical parameter propagation)
        self.params.add_proxy("dp_w", self.generators.dp.params.w)
        …
    def param_check(self):  # handle parameter changes and cross-dependencies
        …
    def prepare(self):  # common data for all views
        # e.g. describe circuit structure/topology
        self.instnamespecs.add("DP", self.generators.dp, sch="I_DP", lay="I_DP", bus=None)
        …
    def schematic(self, cv):  # schematic view description
        i_dp = self.instnamespecs.DP.master.instantiate(cv, pos=Dot(0,0), rot=RotationType.R0, …)
        …
    def layout(self, cv):  # layout view description
        # create instances
        i_dp = self.instnamespecs.DP.master.instantiate(cv, …)  # instance of a generated block
        master = self.open_cellview("mylib", "mycell", "layout")
        i_2 = cv.create_instance(master, "I2", parameters=[…])  # instance of an existing (p)cell
        …
        # create template
        tpl = iip.placeroute.PlaceTemplateStreet(ncols=(4,4), route_opt=…)

        # assign instances to the template (can also be done by arguments of the template constructor)
        tpl.assign_elem(pos=(0,0), elem=i_dp)
        tpl.assign_elem(pos=(1,0), elem=i_2)
        …
        # draw to layout view
        tpl.draw(cv, …)
        …
```

Figure 3: Simplified generator code example with template usage in the layout part.
The shown class methods are a selection of the generator API provided by the IIP core.

Another, rather conceptual question of generator development (no matter if it is done manually or by code generation) is how to handle circuit hierarchies. Basically, hierarchy is a concept for managing complexity by sub-dividing the design problem into smaller pieces and also for reducing complexity as common sub-blocks may be reused. The same actually applies to generators: implementation effort is reduced and reusability increased for more regular and well parameterizable circuits. Thus, a generator developer has to decide which building blocks of a circuit should be realized by a separate generator depending on the required structural and geometrical flexibility and potential reuse. For the template approach shown here, this also determines the sub-blocks to be placed in the template cells.

In conventional analog IC design, lower-level blocks such as operational amplifiers are often designed as flat entities with no further sub-hierarchies although they have functional sub-parts such as differential pairs, biasing, switches, mirrors, etc. each with more or less different sizing, electrical and thus also geometrical constraints. If the IIP Creator flow would be applied to such a flat schematic, the template fields would be filled with the instances of the primitive devices, which would then all be routed through the common channel, and the parameter interface would contain a long list of all the device parameters. The user of the resulting generator might choose a place pattern that lets critical devices, such as the two transistors of the differential pair, be placed close to each other. However, the feasibility of such a flat layout style regarding parasitics and mismatch is not yet proven and will be subject of further investigations. For now, we recommend using base-level generators for circuits of this complexity before creating new generators for them using the IIP Creator flow. On the one hand, these basic generators are already available, and on the other hand, they provide flexible layouts, optimized for area and matching, and a set of specialized parameters. For a new design, they might be used from the beginning already at schematic level. In existing flat schematics, basic structures should be replaced by corresponding instances of generated building blocks (see Figure 2).

IV. RESULTS

We implemented a new layout template following the "street" approach described above as part of the IIP generator API and integrated it into the IIP Creator. An existing OTA (operational transconductance amplifier) design in a 28 nm bulk technology was selected as evaluation example. It was part of a set of LDOs (low drop out) provided by our industry partner [18] and already implemented using the BAG2 [9] generator approach. Figure 4 shows the flat schematic and the generated layout of this source design.

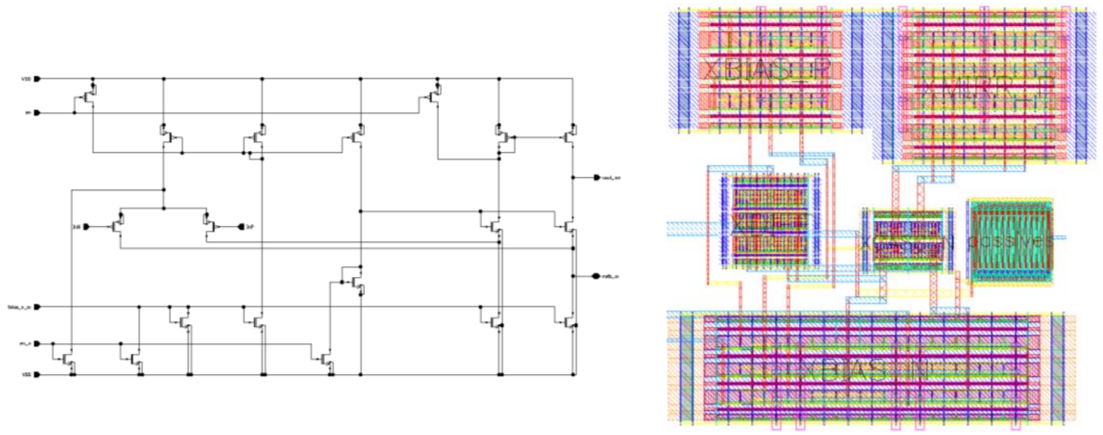
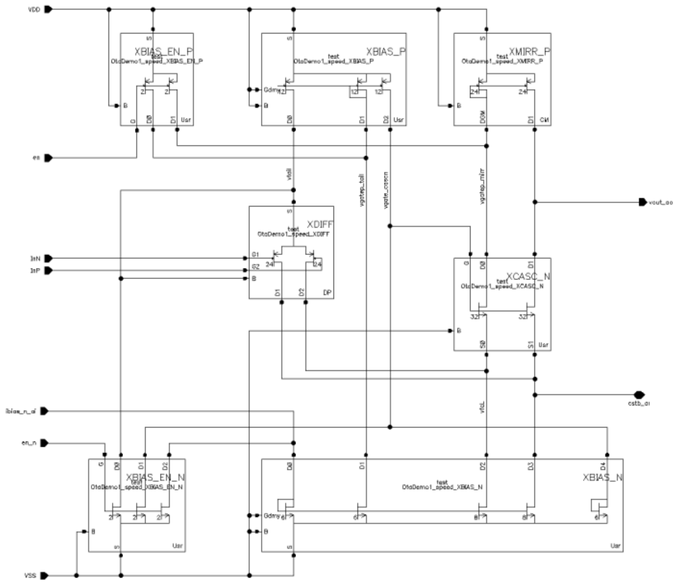
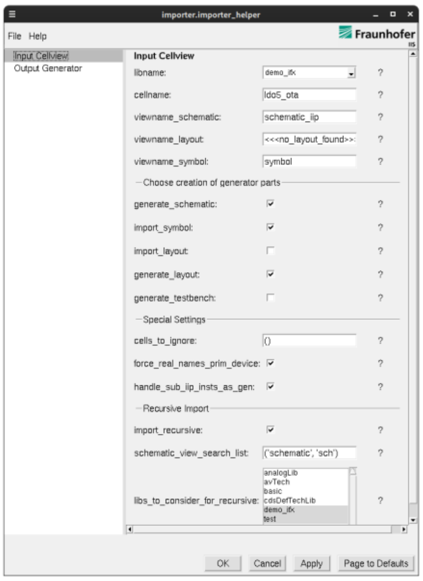


Figure 4: Original OTA design in a 28 nm bulk technology. The layout was generated using the approach in [9][18].

Following the flow depicted in Figure 2, we first identified basic blocks in the original schematic and replaced them by basic IIP generator blocks (Figure 5, left). These basic building blocks largely match the sub-blocks of the original layout: differential pair, biasing for n and p parts, output mirror, and cascode. We did not include the passive devices that were part of the original layout view only. Also, the control switches of the biasing parts were moved to separate blocks because their devices can be very small and thus don't need to be part of the biasing arrays with much larger devices. Using the IIP Creator (Figure 5, right), a new generator for the OTA was created within seconds based on the revised schematic and symbol views. It initially has parameters for all the sub-blocks and for the template-based placement (Figure 6, right).

Additionally, we identified four exemplary sizing cases (power, gain, stability, speed) by pre-layout simulations of the OTA and included them as selectable parameter sets into the generator (Figure 6, left). This step is not necessary for the new generator to work. It rather demonstrates how the generated code may be further specialized to provide a convenient way to switch among pre-defined variants without the need to change several sub-block parameters in very detail (which is still possible, too). Figure 7 shows generated layouts of the OTA in these four sizing variants and each with three different aspect ratios. Generating all views of one variant takes about 30 seconds. The execution time also depends on the performance of the PDK PCells and can differ significantly between PDKs. Currently, adjusting the aspect ratio requires changing several parameters of the template and the sub-blocks. This way, the aspect ratio can be changed between about 1.3 and 5.6 over all variants. There is a maximum difference of about 10 percent in bounding box area among the aspect ratio variants per sizing. Also the influence of the layout parasitics on the circuit performances was investigated by extracted simulations of all the variants. So far we found that the 3 dB bandwidth does not degrade more than 15 percent compared to schematic simulations with a variation of only 3 percent across the generated layout variants of the "speed" sizing. This might be taken as an indication for a low sensitivity of this circuit type to generic and flexible layout styles.




importer.importer_helper

File Help

Input Cellview
Output Generator

Input Cellview

libname: demo_ifx
cellname: ldo5_ota
viewname_schematic: schematic_iip
viewname_layout: <<<no_layout_found>>
viewname_symbol: symbol

Choose creation of generator parts

generate_schematic: ✓
import_symbol: ✓
import_layout: ☐
generate_layout: ✓
generate_testbench: ☐

Special Settings

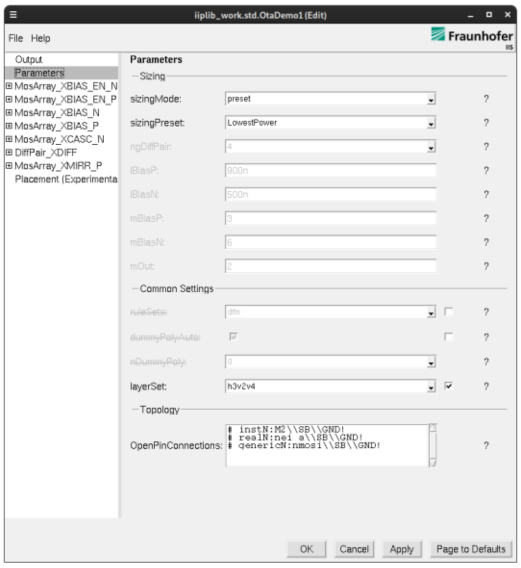
cells_to_ignore: ()
force_real_names_prim_device: ✓
handle_sub_iip_insts_as_gen: ✓

Recursive Import

import_recursive: ✓
schematic_view_search_list: ('schematic', 'sch')
libs_to_consider_for_recursive: analogLib, avTech, basic, cdsDefTechLib, demo_ifx, test

OK Cancel Apply Page to Defaults

Figure 5: The same schematic as in Figure 4 with basic building blocks replaced by blocks from the IIP generator library (left) and the graphical user interface of the IIP Creator that creates executable generator code from existing schematic data (right).


iiplib_work.std.OtaDemo1 (Edit)

File Help

Output
Parameters
⊞ MosArray_XBIAS_EN_N
⊞ MosArray_XBIAS_EN_P
⊞ MosArray_XBIAS_N
⊞ MosArray_XBIAS_P
⊞ MosArray_XCASC_N
⊞ DiffPair_XDIFF
⊞ MosArray_XMIRR_P
Placement (Experimenta

Parameters

Sizing

sizingMode: preset
sizingPreset: LowestPower
ngDiffPair: 4
IBiasP: 900n
IBiasN: 500n
mBiasP: 3
mBiasN: 6
mOut: 2

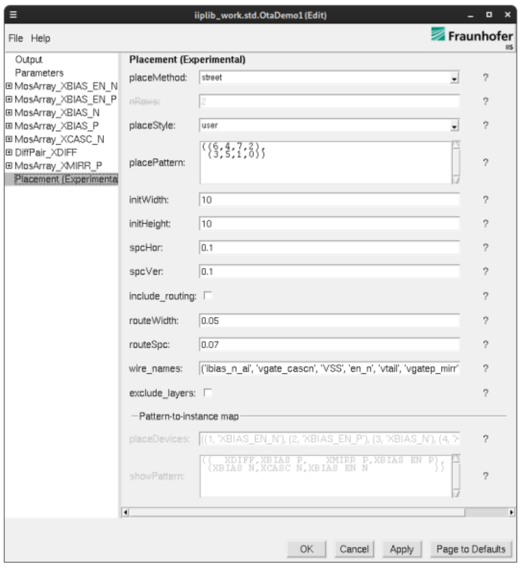
Common Settings

~~ruleSets:~~ dfm ☐
~~dummyPolyAuto:~~ ✓ ☐
~~nDummyPoly:~~ 0
layerSet: h3v2v4 ✓

Topology

OpenPinConnections:
```
# instN:M2\\SB\\GND!
# realN:nei a\\SB\\GND!
# genericN:nmosi\\SB\\GND!
```

OK Cancel Apply Page to Defaults


iiplib_work.std.OtaDemo1 (Edit)

File Help

Output
Parameters
⊞ MosArray_XBIAS_EN_N
⊞ MosArray_XBIAS_EN_P
⊞ MosArray_XBIAS_N
⊞ MosArray_XBIAS_P
⊞ MosArray_XCASC_N
⊞ DiffPair_XDIFF
⊞ MosArray_XMIRR_P
Placement (Experimenta

Placement (Experimental)

placeMethod: street
~~nRows:~~ 2
placeStyle: user
placePattern:
```
((6,4,7,2),
 (3,5,1,0))
```
initWidth: 10
initHeight: 10
spcHor: 0.1
spcVer: 0.1
include_routing: ☐
routeWidth: 0.05
routeSpc: 0.07
wire_names: ('ibias_n_ai', 'vgate_cascn', 'VSS', 'en_n', 'vtail', 'vgatep_mirr'
exclude_layers: ☐

Pattern-to-instance map

placeDevices: ((1, 'XBIAS_EN_N'), (2, 'XBIAS_EN_P'), (3, 'XBIAS_N'), (4, 'X
showPattern:
```
((  XDIFF,XBIAS P,   XMIRR P,XBIAS EN P),
 (XBIAS N,XCASC N,XBIAS EN N            ))
```

OK Cancel Apply Page to Defaults

Figure 6: Graphical user interface of the automatically created generator of the OTA with sizing and other common parameters (left) and placement and template parameters (right). (The preset sizing parameters were added manually after code generation for more convenience.)

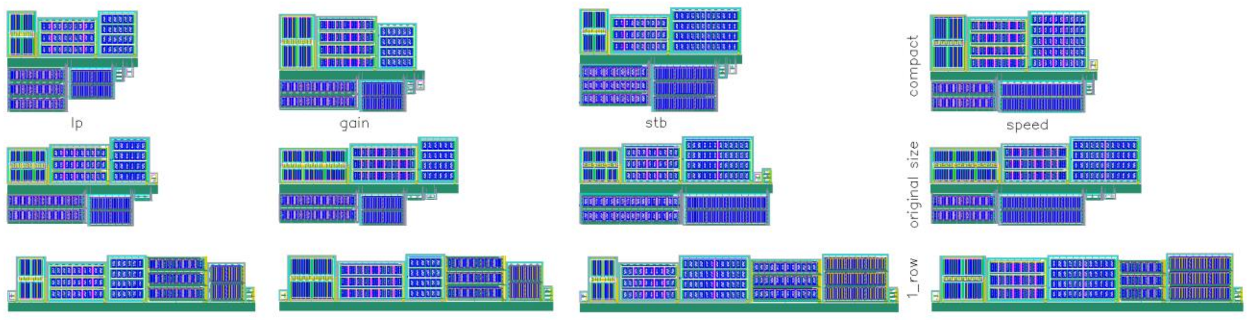


Figure 7: Generated layout variants of the OTA using the selected "street" template, the columns differ by sizing for the pre-calculated performance variants low power, high gain, stability, and high speed while the rows differ by their targeted aspect ratio and placement.

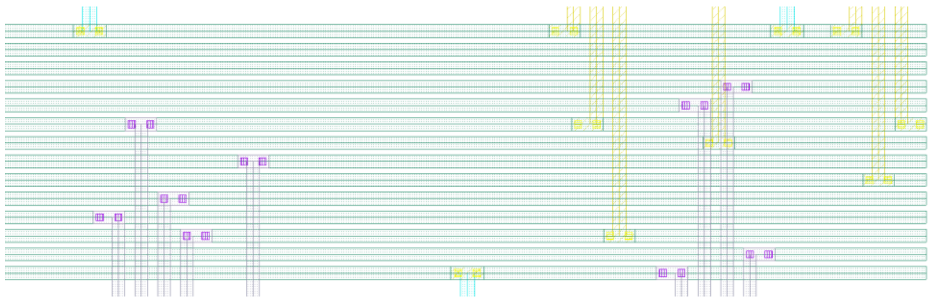


Figure 8: Pragmatic routing channel of the "street" template of one of the examples. In order to avoid conflicts, the layers of horizontal and vertical connections can be ordered such that the horizontal wires use a layer in between the others in the metal stack. This also requires propagating parameters to the sub-generators to adjust appropriate layers and pin positions of the generated instances.

V. Conclusion and Outlook

The automatic creation of template-based generators may close the gap between generators for basic building blocks and designer-driven layout reuse and synthesis for more complex circuits. We see large potential benefits of the IIP Creator approach for both design teams and generator developers. It allows a fast transfer of existing design IP into a much more generic and reusable format. Libraries of circuit generators might define reference IP and design guidelines much better than a long list of documents and legacy designs. For the EDA community the code generation may be a chance to drive a common standard of a generator language.

The demonstrated example also exposes a set of open points that requires further efforts. Template parameters such as place patterns, aspect ratios, and routing styles must better interact between several generators across hierarchies such that their effect on top-level regarding area, aspect ratio, or parasitics (especially when it comes to RF applications) is available early as an estimation before generating the actual layout. This will enable fast layout-aware sizing and early optimization of area or aspect ratio. Also, further investigation of the feasibility of more layout styles (i.e. more templates) for several circuit topologies is required – preferably based on standardized benchmark designs.

Acknowledgment

The authors thank Dr. Husni Habal and his colleagues from Infineon Technologies AG for providing the original design used in [18] and many fruitful discussions about generator-based design approaches and tools.

References

Academic generator- and synthesis-based approaches

[1] J. Scheible, "Optimized is Not Always Optimal – The Dilemma of Analog Design Automation," Proc. of the 2022 Int. Symp. on Physical Design (ISPD '22), pp. 151-158, 2022, doi: 10.1145/3505170.3511042.

[2] J. Scheible und J. Lienig, "Automation of Analog IC Layout – Challenges and Solutions," Proc. of Int. Symp. on Physical Design (ISPD'15), pp. 33-40, 2015.

[3] D. Marolt, "Layout Automation in Analog IC Design with Formalized and Nonformalized Expert Knowledge", Dissertation. Stuttgart, 2018.

[4] A. Graupner, R. Jancke und R. Wittmann, "Generator Based Approach for Analog Circuit and Layout Design and Optimization," Design, Automation & Test in Europe Conference & Exhibition, DATE 2011, March 2011, doi: 10.1109/DATE.2011.5763267.

[5] 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," 13th Int. Conf. on Synthesis, Modeling, Analysis and Simulation Methods and Applications to Circuit Design (SMACD 2016), pp. 1-4, 2016.

[6] B. Prautsch, U. Eichler, T. Reich and J. Lienig, "MESH: Explicit and Flexible Generation of Analog Arrays," 2017 14th Int. Conf. on Synthesis, Modeling, Analysis and Simulation Methods and Applications to Circuit Design (SMACD), pp. 1-4, 2017.

[7] B. Prautsch, R. Wittmann, U. Eichler, U. Hatnik und J. Lienig, "Generators, Templates, and Code Generation for Flexible Automation of Array-Style Layouts," Proc. of the Int. Conf. on Synthesis, Modeling, Analysis and Simulation Methods and Applications to Circuit Design (SMACD 2021), pp. 180-183, July 2021.

[8] N. Lourenço, R. Martins und N. Horta, "Layout-aware Sizing of Analog ICs Using Floorplan & Routing Estimates for Parasitic Extraction," Design, Automation & Test in Europe Conf. & Exhibition (DATE), pp. 1156-1161, 2015.

[9] E. Chang et al., "BAG2: A Process-portable Framework for Generator-based AMS Circuit Design," IEEE Custom Integrated Circuits Conf. (CICC), pp. 1-8, 2018, doi: 10.1109/CICC.2018.8357061.

[10] T. Dhar et al., "ALIGN: A System for Automating Analog Layout," IEEE Design & Test, pp. 8-18, April 2021, doi: 10.1109/MDAT.2020.3042177.

[11] B. Xu et al., "MAGICAL: Toward Fully Automated Analog IC Layout Leveraging Human and Machine Intelligence: Invited Paper," 2019 IEEE/ACM International Conference on Computer-Aided Design (ICCAD), 2019, pp. 1-8, doi: 10.1109/ICCAD45719.2019.8942060.

[12] H. Chen et al., "AutoCRAFT: Layout Automation for Custom Circuits in Advanced FinFET Technologies," ACM International Symposium on Physical Design (ISPD), Virtual Event, Canada, Mar. 27-30, 2022.

Commercial EDA Software

[13] Synopsys Custom Compiler, https://www.synopsys.com/implementation-and-signoff/custom-design-platform/custom-compiler.html, retrieved May 2022

[14] Cadence ModGen, https://www.cadence.com/en_US/home/tools/custom-ic-analog-rf-design/layout-design/virtuoso-layout-suite.html, retrieved May 2022

[15] Cadence PCell Designer, https://www.cadence.com/content/dam/cadence-www/global/en_US/documents/services/cadence-vcad-pcell-ds.pdf, retrieved May 2022

[16] Pulsic Animate, https://pulsic.com/animate/, retrieved May 2022

[17] Analog Rails Automatic Place and Route, http://www.analograils.com/, retrieved May 2022

Proprietary company-internal solutions

[18] F. Passerini et al., "ANAGEN: A Methodology for ANAlog Circuit GENeration", presentation at European Solid-state Devices and Circuits Conference (ESSCIRC/ESSDERC), 2021.

[19] Agile Analog Composa, https://www.agileanalog.com/technical-information/composa-methodology

[20] Silicon Technologies ADONIS, https://silicontechnologiesinc.com/read-more-adonis-re-imaging-analog-design/