BAG2: A Process-Portable Framework for Generator-Based AMS Circuit Design

Eric Chang, Jaeduk Han, Woorham Bae, Zhongkai Wang, Nathan Narevsky, Borivoje Nikolić, Elad Alon
University of California, Berkeley

***Abstract*—We present BAG2, a framework for the development of process-portable Analog and Mixed Signal (AMS) circuit generators. Such generators are parametrized design procedures that produce schematics, layouts, and verification testbenches for a circuit given input specifications. This paper expands on previous work by introducing a universal AMS circuit verification framework into BAG2, as well as two new layout engines, XBase and Laygo, that enable development of process-portable layout generators. We have developed various complex circuit generators as driving examples, including a time-interleaved SAR ADC and a SerDes transceiver frontend. Instances of these designs have been produced in a TSMC 16nm FFC process; we however verify our claims of process portability by presenting circuits generated (using a single methodology code-base and only primitives adapted to the specific process) in various technology nodes, including TSMC 28nm, TSMC 16nm, GLOBALFOUNDRIES 45nm RF-SOI, ST 28nm FD-SOI, and GLOBALFOUNDRIES 22nm FDX.**

I. Introduction

As the nature of scaling has shifted due to both technological and economic barriers, innovations in systems and circuits have become increasingly necessary to meet the needs to next generation designs. However, the stringent and unintuitive design rules of advanced multi-patterned processes [1] along with increased interconnect resistance and capacitance due to dimensional scaling severely lengthen the time spent in post-layout verification and limit designers' ability to explore new circuit designs. Moreover, these factors also impose a significant design cost to migrating AMS circuits to advanced processes, which has driven a trend towards heterogeneous implementations that may introduce performance/power bottlenecks compared to monolithic designs.

Many approaches to shorten the analog circuit design cycle have been explored. In [2], circuit sizing is transformed into a geometric programming problem, and a solver is used to compute individual device parameters from performance specifications. In [3], numerical optimization is used to refine device parameters with schematic simulation results to account for discrepancy between actual circuit model and approximated design equations. In [4], the circuit optimization process is further refined by incorporating layout parasitics estimated from an automated place-and-route tool for analog circuits. While schematic sizing is certainly a key piece of the overall analog design process, as mentioned previously, the bottleneck is currently almost entirely in realizing DRC clean layout and performance-optimized post-layout design. Advances in automatic analog placement and routing have been made [5], [6], [7], however designers have been reluctant to adopt these approaches, often because of "aesthetic" issues with the results produced by the tools (where layouts that are "aesthetically correct" are typically more robust to un- or poorly-modeled layout-dependent effects) [8].

To address these concerns, Crossley *et al*. introduced the Berkeley Analog Generator (BAG) framework and advocated for a generator-centric design approach [9]. The crux of this approach is that instead of designing one circuit instance, the designer should capture their methodology as an executable circuit generator that consists of parameterized procedures which can produce schematic, layout, and verification testbenches from input specifications. With these generators, designers can incorporate fully automated design iteration loops on accurate post-layout simulation data in their design procedure. More importantly, such generators can easily produce many circuit instances for similar applications with different specifications, which promotes design reuse and simplifies complex system design.

Two shortcomings limited the reuseability of generators written in the original version of BAG. First, testbenches generated by BAG expected fixed pin interfaces to the device-under-test. This prevents testbench generators from being shared between similar circuits and leads to duplicated code, which is difficult to maintain and debug. Second, layout generators were written by relying on Synopsys' PyCell API [10], which unfortunately is not broadly available for recent technology nodes, thus hindering application of BAG in state-of-the-art technologies.

In this paper, we present BAG2, an evolved and updated version of BAG that enables development of process-portable circuit generators. We implemented a universal AMS circuit verification framework enabled by BAG2's schematic generation API, and incorporated this methodology into the generator-based design flow. We also developed two new layout generation engines, XBase and Laygo, both of which interface with BAG2 to provide process-independent layout APIs for designers to program layout generators. Using BAG, XBase, and Laygo, we have realized complete time-interleaved SAR ADC and SerDes transceiver frontend generators, and produced verified instances using these generators that were taped out in TSMC 16nm. Furthermore, we demonstrate layout process portability by generating various circuit blocks in TSMC 28nm, TSMC 16nm, GLOBALFOUNDRIES 45nm RF SOI, ST 28nm FD-SOI, and GLOBALFOUNDRIES 22nm FDX processes; all of these instances are produced using a single code-base for the methodology combined with process-

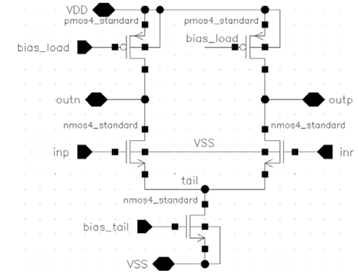
978-1-5386-2483-8/18/$31.00 ©2018 IEEE

```python
class bag_serdes_ec__diffamp(Module):
    def __init__(self, bag_config, parent=None, prj=None, **kwargs):...

    def design(self, lch, w_in, n_in, th_in,
               w_load, n_load, th_load,
               w_tail, n_tail, th_tail):

        self.instances['XIP'].design(w=w_in, l=lch, nf=n_in, intent=th_in)
        self.instances['XIN'].design(w=w_in, l=lch, nf=n_in, intent=th_in)
        self.instances['XLP'].design(w=w_load, l=lch, nf=n_load, intent=th_load)
        self.instances['XLN'].design(w=w_load, l=lch, nf=n_load, intent=th_load)
        self.instances['XT'].design(w=w_tail, l=lch, nf=n_tail, intent=th_tail)
```

(a) Schematic generator.

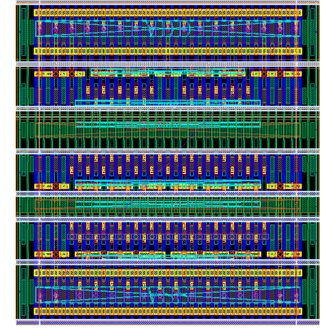


(b) Schematic template.

```python
class DiffAmp(SerdesRXBase):
    def __init__(self, temp_db, lib_name, params, used_names, **kwargs):...
    def draw_layout(self):
        # ...
        self.draw_base(lch, fg_tot, ptap_w, ntap_w, nw_list, nth_list,
                       pw_list, pth_list, ng_tracks=ng_tracks,
                       nds_tracks=nds_tracks, pg_tracks=pg_tracks,
                       pds_tracks=pds_tracks, n_orientations=n_orient,
                       p_orientations=p_orient, **kwargs)

        mp_wires = self.draw_mos_conn('nch', 1, col_inp, n_in, sdir, ddir)
        mn_wires = self.draw_mos_conn('nch', 1, col_inn, n_in, sdir, ddir)
        # ...
        inp_wires = self.connect_to_tracks(mp_wires['g'], wire_layer, inp_tidx)
        inn_wires = self.connect_to_tracks(mn_wires['g'], wire_layer, inp_tidx)
        self.add_pin('INP', inp_wires)
        self.add_pin('INN', inn_wires)
        # ...
```

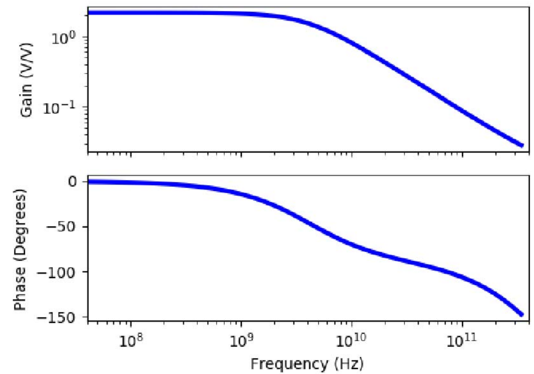
(c) Layout Generator (XBase).



(d) Generated Layout.

```python
def design_diffamp(prj, gain_targ, bw_targ, meas_config):
    done = False
    gain, bw, amp_params = None, None, None
    gain_min, bw_min = gain_targ, bw_targ
    tran_db = get_transistor_database(prj)
    while not done:
        amp_params = compute_params(gain_min, bw_min, tran_db)
        amp_info = generate_instance(prj, amp_params)
        meas = setup_measurement(prj, amp_info, meas_config)
        gain, bw = meas.do_measurement()
        if gain >= gain_targ and bw >= bw_targ:
            done = True
        else:
            if gain < gain_targ:
                scale = gain / amp_params['gain_expected']
                gain_min = gain_min / scale
            if bw < bw_targ:
                scale = bw / amp_params['bw_expected']
                bw_min = bw_min / scale
    print('diffamp gain = %.4g' % gain)
    print('diffamp bw = %.4g' % bw)
```

(e) Design script.



(f) Simulated result.

Fig. 1: Generator design flow example.

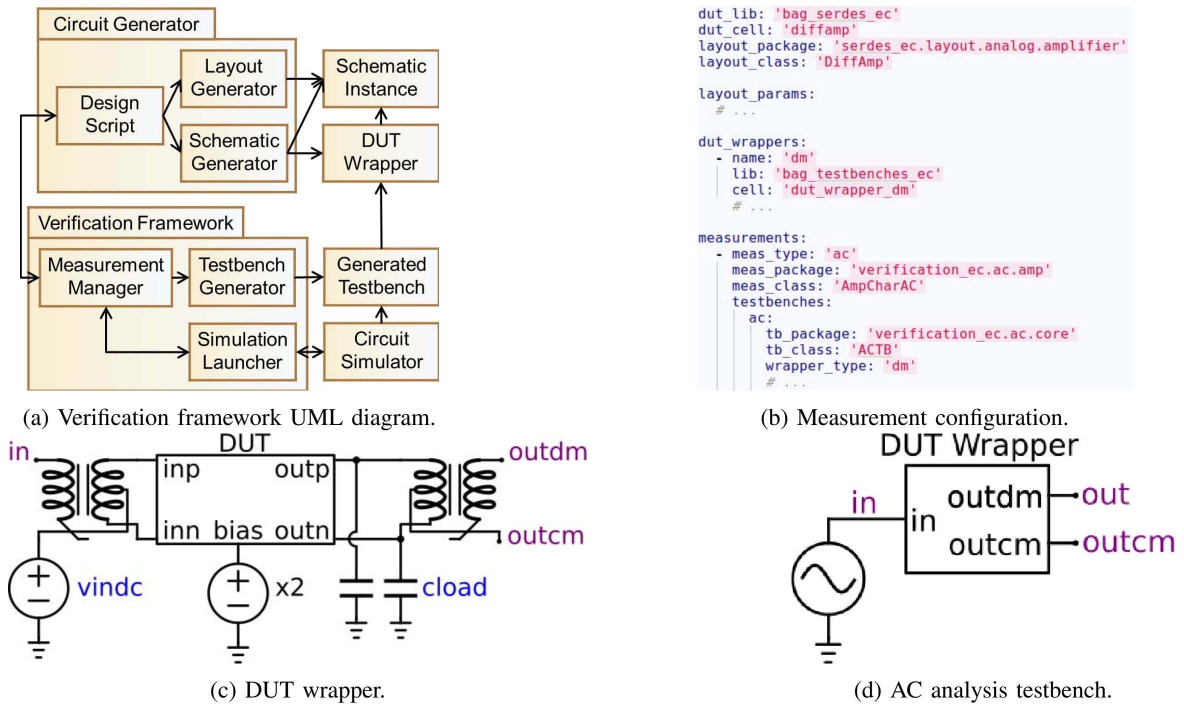
specific primitives.

The rest of the paper is organized as follows. Section II reviews the generator-based design flow using the differential amplifier as a driving example, and Section III then describes the universal AMS circuit verification framework. Section IV presents an overview of the key principles that enable process-portable layout generation. Section IV-A presents the XBase layout engine, which provides various classes and functions for writing passive and analog circuit generators based on parameterized primitives. Section IV-B describes the Laygo layout engine, which is an alternative to XBase that relies on hand-designed layout primitives, and is thus better suited for creating compact custom circuits.

Section V describes generators developed with these layout engines and summarizes our process portability experiments. Finally, we conclude and discuss future work in Section VI.

II. Generator Design Flow

In this section, we walk through an example design process that could be captured within a differential amplifier generator to provide an overview of the BAG2 framework. First, similar to traditional circuit design, the designer creates a normal schematic of the circuit, as shown in Figure 1b. This schematic serves as a template for the schematic generator, and BAG2 will create new circuit instances by copying and modifying this template. Compared to a purely netlist-based approach,



(a) Verification framework UML diagram.

(b) Measurement configuration.

(c) DUT wrapper.

(d) AC analysis testbench.

Fig. 2: Verification framework example.

this allows BAG2 to generate human-readable schematics that integrate well into the traditional manual design flow. After the schematic template is created, the designer chooses the input parameters of the corresponding schematic generator. These parameters are usually low-level structural parameters (such as transistor dimensions) instead of high-level specifications, as this allows for the reuse of the same schematic generator for different high-level design specifications. With the input parameters of the schematic generator determined, the designer implements the schematic generation routine in a Python class (Figure 1a), which calls various BAG2 functions to transform the schematic template into a specific circuit instance. In addition to modifying device dimensions, BAG2 provides an API for adding or modifying pins, instances, and instance connections. This allows the designer to freely modify the underlying circuit structure, such as adding cascode transistors, creating variable array structures, and so on.

After the schematic generator is implemented, the designer develops the corresponding layout generator using either XBase or Laygo (Figure 1c), described in Section IV-A and Section IV-B, respectively. The inputs to the layout generator are generally a super-set of the inputs to the schematic generator, as layout parameters such as aspect ratio and signal wire width/spacing are not necessary for schematic generation. After the layout generator is complete, a designer can use BAG2 to generate schematic and layout instances with various parameter combinations and verify that the generators produce DRC and LVS clean layouts (Figure 1d).

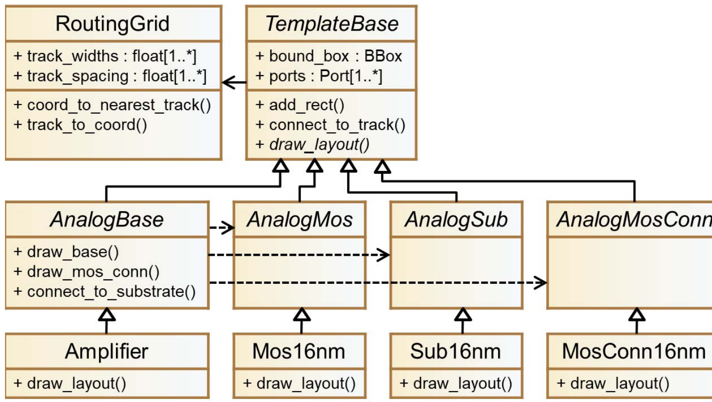
Next, the designer codes the design procedure into a design script (Figure 1e) that takes in top-level specifications and computes parameters for the schematic and layout generator. Typically, the design procedure uses design equations along with a pre-characterized technology database to compute an initial starting point (such as the g_m/I_D sizing methodology [11]), then iterates on post-extraction simulation results to account for layout parasitics and converge to the final solution.

To run simulations and get circuit performance from simulation data, the designer simply specifies the measurement parameters in a configuration file (Figure 2b), and passes this configuration file to the universal verification framework. The verification framework will then setup testbenches, simulate given circuit instances, and report circuit performance (Figure 1f). The implementation of the verification framework is described in Section III.

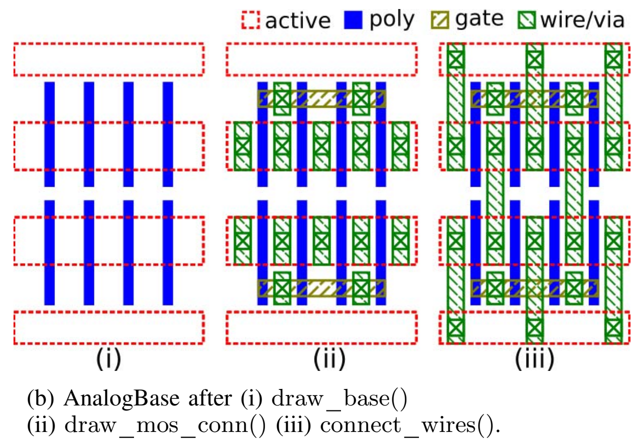
Using this design flow, especially if the designer avoids hard-coding parameters in the design script and various generators, the design process of any circuit block can be easily repeated to achieve optimal performance (or at least performance in line with what the designer themselves would achieve if they repeated the same methodology) with different specifications or in another technology node. In Section V-A we present post-layout extracted results of differential amplifiers generated using this flow in various technologies.

III. Universal Verification Framework

In traditional analog circuit design, verification testbench and simulation data processing are often tightly coupled to the specific circuit and technology node, which limits testbench reusability and prevents development of a verification standard for analog circuits. Previous work has addressed this problem by defining clear interfaces between circuits and tests that



(a) XBase top-level UML diagram.



(b) AnalogBase after (i) draw_base() (ii) draw_mos_conn() (iii) connect_wires().

Fig. 3: XBase class hierarchy and example layout.

operate on them, then building a hierarchical repository of re-useable test components that enables the design of circuit-independent tests [12]. We incorporated the same principle and developed a universal verification framework using BAG2's schematic generation API. Figure 2a outlines the interfaces between design script, verification framework, and various generators.

We walk through the verification flow by continuing the differential amplifier example in Section II. First, before invoking the verification framework, the design script generates device-under-test (DUT) wrapper schematics (Figure 2c) for each testbench it is going to instantiate. The DUT wrapper serves two main purposes. First, it transforms the DUT input/output pin interface to a standard interface defined by the testbench. In Figure 2c, it instantiates two ideal baluns to perform differential-to-single-ended conversion, and adds various bias voltage sources. Second, it provides proper loading to the DUT, which can be capacitors, resistors, or even other circuit instances.

After DUT wrappers are generated, the verification framework instantiates one measurement manager for each measurement specified in the configuration file (Figure 2b). Each measurement manager instantiates and simulates one or more testbenches, and post-processes simulation data to determine circuit performance. In this example, the measurement manager simply instantiates an AC analysis testbench and then compute amplifier gain and bandwidth from the Bode plot. After all measurements are finished, the results are reported back to the design script, which are used to fine tune circuit parameters if the circuit does not meet specifications.

With this approach, a universal verification standard can be defined for all analog circuits. Complex verification procedures simply need to be programmed once by experienced designers and can be applied to all similar circuits.

Having provided an overview of the flow and verification framework, since layout is the key bottleneck to address in analog design productivity, we move on to our new layout generation API that enables the development of truly process-portable circuit generators.

IV. Process-Portable Layout

There are two key insights behind our solution for process-portable layout generators. First, despite vast differences in the device geometry and design rules, the layout floorplan for a circuit used in a given context/application almost always has many largely invariant characteristics. Figure 3b shows a typical floorplan for an amplifier circuit. By using vertical wires to connect transistor and substrate rows, the number of wires naturally scales with number of fingers, thus minimizing interconnect resistance and maximizing electromigration-limited transistor current density. Therefore, by constructing a layout API that allows designers to draw various common floorplans and abstract away geometry details of the underlying devices, generators written in this API can be process-portable.

Second, complex and un-intuitive multi-patterning design rules have driven circuit and mask designers to enforce regular routing grids even in custom layouts. Therefore, by enforcing a routing grid system where both the width and spacing of the wires are quantized, and wires on the same layer must travel in the same direction (with adjacent layers having alternating directions), we can ensure DRC correctness by simply adjusting the routing grid parameters. Most importantly, layout generators written to draw all wires on routing grid can be easily ported across different process nodes. As an added benefit, for multi-patterned technology nodes where low-level metals must be colored explicitly by the user, the coloring can be automatically done based on the parity of the routing track index.

In the next two sections, we present two specific layout engines that implement these concepts.

A. *XBase Overview*

The top-level UML diagram for XBase is shown in Figure 3a. Python abstract base classes are used to encapsulate common layout methods and separate process agnostic generators from process-specific primitives.

TemplateBase is the fundamental class that all layout generators are based on. It defines two sets of APIs, one high-level and one low-level, that designers call to create layout geometries. The low-level API contains methods such as add_rect() and add_via() that allow a designer to draw any primitive geometry without any restrictions. These methods do not guarantee DRC correctness, and therefore are mostly used only to implement the primitives. The high-level API enforces the gridded routing methodology, and contains methods such as connect_to_track() and connect_wires() that create wires on the routing grid described by the RoutingGrid class. Since they produce layouts that adhere to (adjustable) grids, Layout generators written solely using methods in the high-level API are highly process-portable. Finally, TemplateBase defines an abstract method called draw_layout(), which must be implemented by its subclasses to create layout for the specific circuits.

Each type of layout floorplan is implemented as an abstract subclass of TemplateBase, which provide methods for drawing the specific floorplan. AnalogBase is an abstract class specialized in drawing layout floorplans for electromigration constrained analog circuits, as described in Section IV. AnalogBase provides the draw_base() method, which draws rows of transistors and substrate taps based on user supplied parameters, such as number of fingers per row, number of rows, number of horizontal routing tracks per row, etc. It also defines draw_mos_conn() and connect_to_substrate(), which connects transistors and substrate taps to the routing grid. To create an amplifier generator with this floorplan (represented by Amplifier class in Figure 3a), designers simply sub-class AnalogBase and implement the draw_layout() method by calling draw_base() and other related methods with proper parameters.

To enable process portability, abstract primitive classes are used to separate process-specific layouts from AnalogBase. These primitive classes define abstract methods that encapsulate all design rules relevant to the floorplan they support. Methods in AnalogBase such as draw_base() call these functions and determine how to correctly assemble the primitives together. As a result, by simply changing routing grid parameters and implementations of these abstract primitive classes, the designer can generate circuits in another process technology using the same layout generator.

In addition to AnalogBase, XBase also provides support for a number of layout styles, including ResArrayBase for drawing resistor arrays, StdCellBase for black-boxing digital standard cells, and custom function to create MOM capacitors out of parallel wires (which are drawn on a special routing grid to maximize capacitor). With these classes and functions, many AMS circuit generators can be implemented.

B. Laygo Overview

Laygo stands for LAYout with Gridded Objects, and as the name implies, it was developed for generating DRC clean-by-construction layouts by placing layout elements on grids and utilizing relative and symbolic information (rather than hard-coded numbers) to place them. The main difference between Laygo from XBase is that Laygo uses manually designed, process-specific layout primitives that are not in and of themselves programmable, but can hence be heavily customized. In contrast, XBase uses abstract base classes to support programmable primitives, which requires a larger level of effort to realize the same degree of (process-specific) customization but retain programmability while adhering to the API.

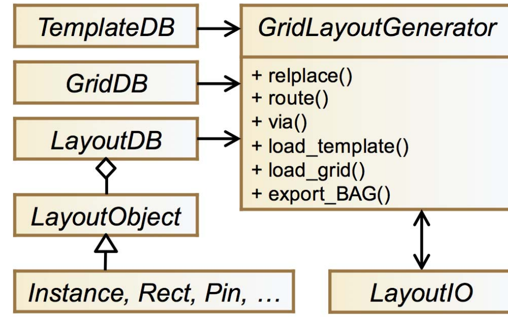


Fig. 4: Laygo structure.

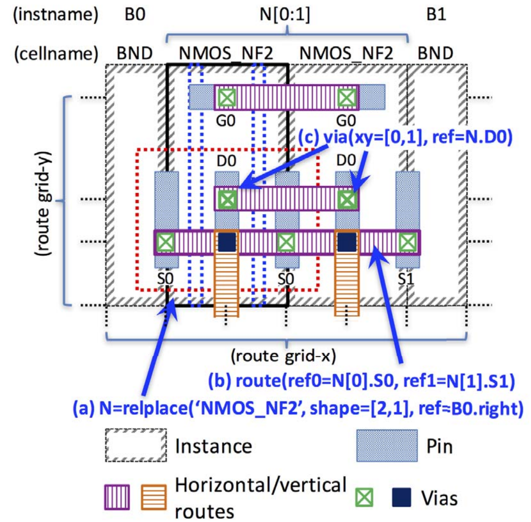
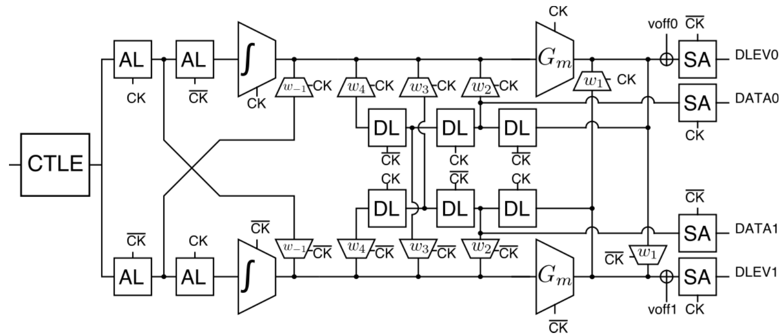


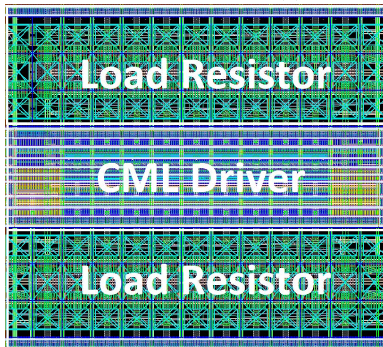
Fig. 5: Laygo generation example (a) relplace (b) route (c) via.

Figure 4 shows the structure of the Laygo framework. GridLayoutGenerator is the main class for layout generation. Users create various layout geometries by calling functions defined in GridLayoutGenerator, such as relplace() (for instance placements), route() (for wire routing), and via() (for via placement).

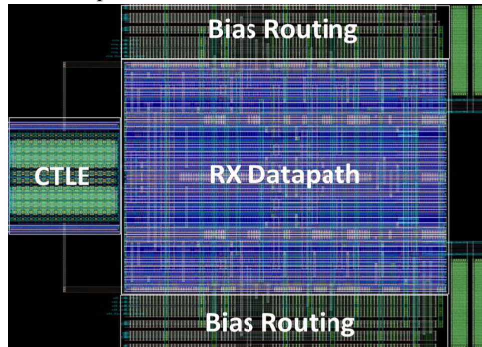
The relplace() function places instances (whose generic parameters are stored in the TemplateDB object) based on integer grid coordinates (within the placement grid) and relative information with neighboring structures (ref parameter in Figure 5(a)). The GridDB object stores the placement grid information (x, y resolution) for the placement operation. Since the relplace() function uses an abstract grid as well



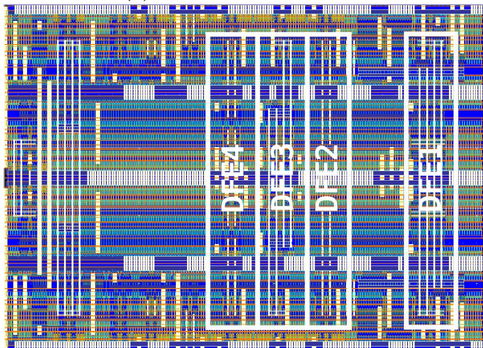
(a) Receiver frontend block diagram with 4-tap DFE.



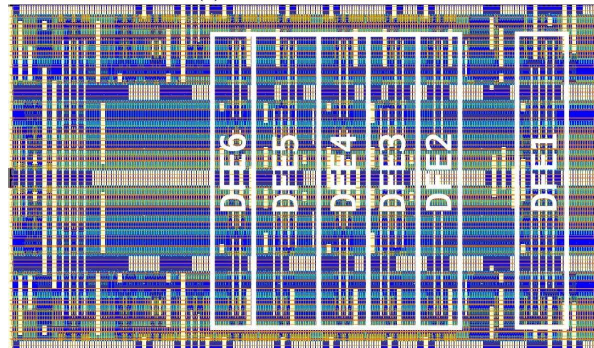
(b) Transmitter frontend.



(c) Receiver frontend.



(d) Receiver datapath with 4-tap DFE, 36µm by 26µm.



(e) Receiver datapath with 6-tap DFE, 44µm by 26µm.

Fig. 6: SerDes frontend block diagram and layout photo.

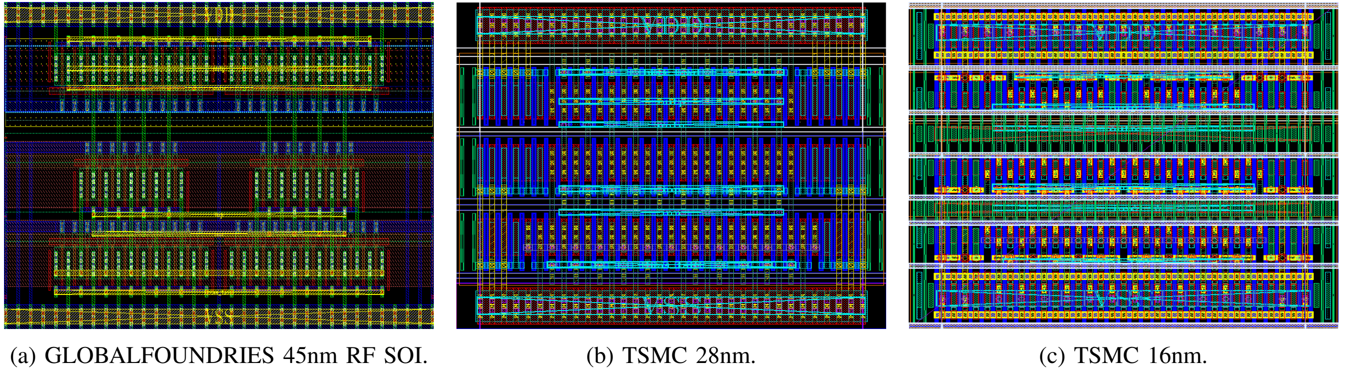
as information about instances relative to each other, process portability is easily achieved if all primitive templates are designed to be aligned with the process-specific placement grids. The $\mathrm{route}()$ function draws routing patterns on predefined routing grids. Similar to the $\mathrm{relplace}()$ function, in order to create wires without dealing with physical coordinates, the $\mathrm{route}()$ function receives integer coordinates (within the abstract routing grids) and/or target objects to connect to the routing wires (Figure 5(b)). All integer routing coordinates are converted to physical coordinates by combining the integer values and routing grid information (pitches, widths). The $\mathrm{via}()$ function places via objects, using a similar method with the aforementioned functions (Figure 5(c)).

These three placement and routing functions compose the core generator flow, but high-level functions that construct more complex geometries (by combining the core functions) are provided as well. It is important to note that although the layout is gridded, the grids need not necessarily be uniform within a given layer. For example, one may want to allow low-level horizontal routing tracks to connect to the gate, source, and drain of the transistors, which may not necessarily be spaced uniformly. As shown in Figure 5), Laygo supports such non-uniform grids (as long as the grid can be specified by a finite list of repeating widths/spaces), which enables high quality-of-results (by optimizing the grid values for a given technology) while still abstracting the design rules.

All layout structures are stored in $\mathrm{LayoutDB}$ and exported to $\mathrm{LayoutIO}$, to create the final layout in OpenAccess or GDS format.

V. Results

Using BAG2, we have designed a number of generators for various functional blocks, and exercised these generators



(a) GLOBALFOUNDRIES 45nm RF SOI. (b) TSMC 28nm. (c) TSMC 16nm.

Fig. 7: Generated differential amplifier in various processes. Images not to scale.

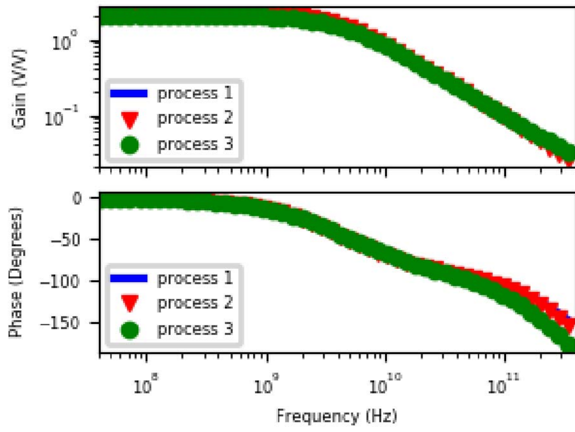


Fig. 8: Bode plot of differential amplifiers.

across multiple process technologies. Specifically, we have implemented a SerDes frontend layout generator using XBase (Figure 6). The generator supports a simple CML driver as the transmitter, and a receiver datapath employing the dynamic latch and charge integration topology [13], which includes passive CTLE, FFE, and DFE, shown in Figure 6a. All transistor sizes are programmable, as is the number of DFE taps. Two instances of the receiver datapath with 4-tap and 6-tap DFE are shown in Figure 6d and Figure 6e, respectively. A CTLE/1-tap FFE/4-tap DFE instance produced by this generator was taped out in TSMC's 16nm FFC process; this instance was verified in post-layout extracted simulations to support 20 Gb/s. This generated Serdes frontend instance occupies an area of 195µm by 110µm, including AC coupling capacitors and ESD diodes.

We further developed a time-interleaved SAR ADC generator using both Laygo and XBase (Figure 9). The generator consists of clock distribution and calibration circuitry (Laygo), frontend samplers (XBase), reference DACs and buffers (XBase), SAR ADC cores (Laygo), and an output retimer (XBase). The ADC top-level specifications are all programmable, and an instance targeting 9.6 GS/s with 8-way interleaving, 6 bits of ENOB with 9 physical output bits was also taped out in TSMC's 16nm FFC process. The total area of the entire generated ADC instance is 500µm by 100µm.

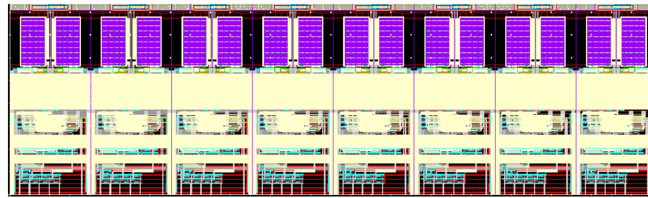
A. *Process Portability*

To validate our claim of process portability, we ran the differential amplifier generator presented in Section II (with layout generators written in XBase) in GLOBALFOUNDRIES RF SOI 45nm, TSMC 28nm, and TSMC 16nm. All of the amplifiers are designed to drive a 100 fF capacitive load with a target DC gain of 2, a target 3-dB bandwidth of 4 GHz, and minimum power consumption. The generated layouts are DRC-clean, shown in Figure 7, and the post-layout extraction simulated Bode plots are shown in Figure 8. The final designs have DC gains of 2.07, 2.22, and 2.17, and 3-db bandwidths of 4.12 GHz, 4.15 GHz, and 4.03 GHz. Despite vast differences in design rules and underlying transistor device physics, the generator was able to produce DRC/LVS clean circuits that closely met the target specifications.

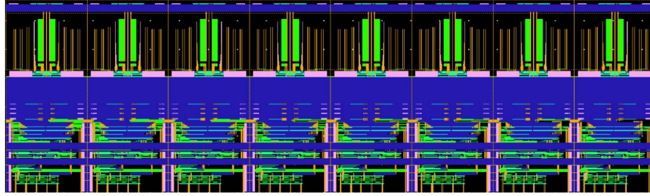
To demonstrate layout portability of more complex blocks, Figure 9 shows the generated ADC core layout in TSMC 16nm, ST 28nm FD-SOI, and GLOBALFOUNDRIES 22nm FDX processes. The ADC core consists of sense amplifiers, capacitor banks, and SAR logic. The generated layouts are all DRC and LVS clean without any manual modification.

VI. Conclusion

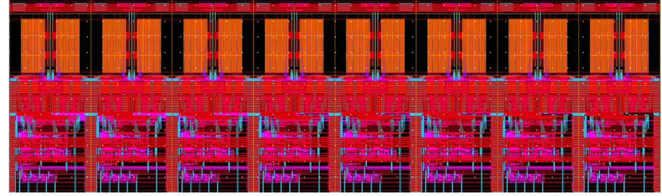
A substantial change in AMS circuit design and verification flows is necessary to continue innovation beyond Moore's law scaling. In this paper we introduce BAG2 as a new framework to support generator-based design flows, where designers capture their knowledge in the form of executable generators that can then be re-used to realize designs with varying specifications and process technologies. BAG2 expands on our previous work by including a universal verification framework built with testbench generators, thus enabling improved reuse and the potential introduction of a verification standard for AMS circuits. BAG2 also includes two new layout generation frameworks - XBase and Laygo - both of which explicitly address process portability and the challenges associated with advanced process technologies by enforcing (programmable)



(a) TISAR ADC core in TSMC 16nm.



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



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

Fig. 9: TISAR ADC layout photos. Images not to scale.

gridding. To prove the viability of this approach, we implemented multiple complex generators of a number of circuit blocks - including a SerDes frontend and a time-inteleaved SAR ADC - and taped out generated instances for these blocks in TSMC's 16nm FFC process. We further demonstrated process portability by using the same generator code (but process-specific primitives) to create DRC/LVS clean instances in a variety of technology nodes (including bulk, FinFET, PD- and FD-SOI) with widely varying rules and geometries.

We believe that these results show that a generator-based design methodology is a viable path to enhance the productivity of AMS circuit design. Moving forward, beyond expanding the library of available generators, we believe that complete SOC generators will be developed by integrating BAG with digital generator approaches, and in particular with hardware construction languages such as Chisel [14]. BAG [15], XBase [16], and Laygo [17] have all been released under open source licenses, and preliminary documentation as well as tutorials are currently available (as well as being under continuing and active development).

Acknowledgment

This work was funded in part by the DARPA CRAFT program (HR0011-16-C-0052), and the authors wish to acknowledge the contributions of their collaborators on this effort, Cadence Design Systems and Northrop Grumman Corporation. The authors would like to further acknowledge the sponsors, students, and faculty of the Berkeley Wireless Research Center.

References

[1] M. White. (2017) Are you (really) ready for your next node? [Online]. Available: http://www.electronicdesign.com/eda/are-you-really-ready-your-next-node

[2] M. del M. Hershenson, S. P. Boyd, and T. H. Lee, "Optimal design of a CMOS op-amp via geometric programming," in *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 20, no. 1, Jan. 2001, pp. 1–21.

[3] A. Vladimirescu, R. Zlatanovici, and P. Jespers, "Analog circuit synthesis using standard EDA tools," in *IEEE International Symposium on Circuits and Systems (ISCAS)*, May 2006, pp. 5239–5243.

[4] G. Zhang *et al.*, "A synthesis flow toward fast parasitic closure for radio frequency integrated circuits," in *Design Automation Conference (DAC)*, Jun. 2004, pp. 155–158.

[5] L. E. Henrickson and E. S. Petrus, "Sytem and method for utilizing meta-cells," U.S. Patent 7 587 694B1, Sep. 8, 2009.

[6] Cadence Design Systems. (2016) Virtuoso Layout Suite GXL. [Online]. Available: https://www.cadence.com

[7] Synopsys. (2016) Custom Compiler. [Online]. Available: https://www.synopsys.com

[8] R. A. Rutenbar, "Analog circuit and layout synthesis revisited," invited talk, at ACM Int'l Symposium on Physical Design, Mar. 2015. [Online]. Available: http://www.ispd.cc/slides/2015/keynote_Rutenbar.pptx

[9] J. Crossley *et al.*, "BAG: A designer-oriented integrated framework for the development of AMS circuit generators," in *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE, Nov. 2013, pp. 74–81.

[10] S. Alassi and B. Winter, "PyCells for an open semiconductor industry," *CoRR*, 2016. [Online]. Available: http://arxiv.org/abs/1607.00859

[11] F. Silveira, D. Flandre, and P. G. A. 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," *IEEE J. Solid-State Circuits*, vol. 31, no. 9, pp. 1314–1319, 1996.

[12] J. Mao, "CircuitBook: A framework for analog design reuse," Ph.D. dissertation, Stanford University, May 2013.

[13] J. Han *et al.*, "Design techniques for a 60 Gb/s 173 mW wireline receiver frontend in 65 nm CMOS technology," *IEEE J. Solid-State Circuits*, vol. 51, no. 4, pp. 871–880, April 2016.

[14] J. Bachrach *et al.*, "Chisel: Constructing hardware in a Scala embedded language," in *Design Automation Conference (DAC)*, San Francisco, CA, USA, June 2012.

[15] Berkeley analog generator, main framework. [Online]. Available: https://github.com/ucb-art/BAG_framework

[16] Berkeley analog generator, XBase. [Online]. Available: https://github.com/ucb-art/BAG2_TEMPLATES_EC

[17] Berkeley analog generator, layout with gridded objects (Laygo). [Online]. Available: https://github.com/ucb-art/laygo