BAG: A Designer-Oriented Integrated Framework for the Development of AMS Circuit Generators

(Invited Designer Track Paper)

J. Crossley, A. Puggelli, H.-P. Le, B. Yang, R. Nancollas, K. Jung, L. Kong,
N. Narevsky, Y. Lu, N. Sutardja, E. J. An, A. L. Sangiovanni-Vincentelli, E. Alon

Department of Electrical Engineering and Computer Science, University of California, Berkeley
Email: {crossley, puggelli, phucle, byang, racheln, kwangmo, konglk,
narevsky, yuelu, nsutardj, eunjian, alberto, elad}@eecs.berkeley.edu

***Abstract*—We introduce BAG, the Berkeley Analog Generator, an integrated framework for the development of generators of Analog and Mixed Signal (AMS) circuits. Such generators are parameterized design procedures that produce sized schematics and correct layouts optimized to meet a set of input specifications. BAG extends previous work by implementing interfaces to integrate all steps of the design flow into a single environment and by providing helper classes – both at the schematic and layout level – to aid the designer in developing truly parameterized and technology-independent circuit generators. This simplifies the codification of common tasks including technology characterization, schematic and testbench translation, simulator interfacing, physical verification and extraction, and parameterized layout creation for common *styles* of layout. We believe that this approach will foster design reuse, ease technology migration, and shorten time-to-market, while remaining close to the classical design flow to ease adoption. We have used BAG to design generators for several circuits, including a Voltage Controlled Oscillator (VCO) and a Switched-Capacitor (SC) voltage regulator in a CMOS 65nm process. We also present results from automatic migration of our designs to a 40nm process.**

I. Introduction

The recent trend in embedding multiple applications into a single System-on-Chip (SoC) has resulted in a substantial increase in the number of digital and Analog/Mixed-Signal (AMS) components integrated per die [1]. Although most functionality in such integrated systems is implemented with digital circuitry, some functions are intrinsically analog, and analog circuits are needed to guarantee system functionality and performance (e.g. radio transceivers, temperature sensors, voltage regulators). Although these AMS components typically occupy a small fraction of the whole IC, they often represent a bottleneck in terms of design time. This is mainly due to the lack of well-defined steps in the design flow and of the ability to capture them in an executable way. As a result, analog designers are typically involved vertically in the design process, from the definition of system specifications down to layout implementation, and multiple iterations across all layers are required to achieve the desired performance. This is even more true today in designs started or migrated into advanced technology nodes, where the layout has a dramatic impact on the performance of the components, and its effects must be taken into consideration early in the design process. It would thus be desirable to automate the design of AMS circuits and foster their reuse across multiple SoCs and technology generations, to shorten time-to-market of new products and to free analog designers from performing repetitive tasks (e.g. circuit redesign due to a change in the specifications or technology migration).

A typical custom design flow for AMS circuits can be roughly divided in two phases. The functional-description phase starts from the translation of an application concept into a system functional model, using High-level tools like Matlab, Python or Simulink. Next, each system functional block is mapped to a specific circuit architecture and application specifications are translated into performance constraints for each architectural block. This marks the beginning of the architecture-design phase, in which the circuit representation of each sub-cell within the architecture gets refined from its functional description down to transistor level. Finally, all component cell layouts are placed and their interconnections are routed. Each step in both phases is followed by extensive verification to guarantee that system functionality and performance are preserved, and iterations are often needed across all layers to compensate for unforeseen affects.

Several techniques to automate various steps of this flow have been explored. They can be divided into two categories [2]. In "Knowledge-Based Techniques" [3], [4], design steps tailored to specific circuit architectures (e.g. Flash ADC, Switched-Capacitor filters, etc.) are encoded into a design plan, i.e., procedural scripts that mimic the designer activity. These scripts are usually fast to run, as well as serving as functional documentation of the design, and the designer maintains full control of the synthesis flow, thus easing modifications and debugging. On the other hand, the activity of setting up the synthesis scripts might be long and error prone, and new scripts are needed for each new design, so a *library* of design plans is required to make these approaches widely-adoptable. "Optimization-Based Techniques" [5], [6], instead, keep the functional description of the design under analysis separated from a *library* of available architectural implementations. The user enters the desired system functionality in terms of behavioral models and performance constraints. Synthesis is then cast into one or more optimization problems, where the user-defined cost function and constraints drive the tool to select and size the library architecture that optimally meets the specifications. These constraint-driven approaches have been shown to produce high-performing designs, and can, in principle, seamlessly operate on a large class of AMS circuits using the same design steps. On the other hand, they can have long runtimes due to the large design space to be explored for practical analog circuits (> 100 devices), and can still require substantial design experience to limit the degrees of freedom to be explored. Further, the returned circuit implementation might be difficult to debug or modify, since the tool acts as a black-box to the user, preventing them from building intuition on how the design has been generated.

Despite this effort, the designer community has been reticent to widely adopt automation software, remaining anchored to the highly manual nature of the custom flow. We believe that historically this reluctance has stemmed from the lingering difference in perspective between the two communities. The CAD community has mainly focused in developing modeling frameworks to capture system functionality and optimization algorithms for architecture selection and sizing, but it has left to the designers the burden of creating a library of architectures compatible with the proposed frameworks. The designer community, instead, has lamented the excessive effort required to initially set up a new (automated) design flow, and has shown skepticism towards automation, motivated both by the belief that a better design can be obtained through a manual effort and by the fear of losing the central role (and the job) in the design process.

This situation is now rapidly changing. The need to integrate an increasing number of AMS circuits per chip has pushed the designer community to an inflection point with regards to design automation, since: 1) the ability to quickly redesign a block now outweighs the initial effort to set up an automated synthesis flow; 2) there is a concrete request to create more designs with the same number of people, instead of the same number of designs with fewer people, and; 3) the efforts of almost three decades of research have greatly improved analog CAD tools in terms of ease of use and quality of the synthesized designs. In fact, CAD research in the field is still active,

978-1-4799-1071-7/13/$31.00 ©2013 IEEE

both in the academic and industrial worlds [7]–[10]. In the following, we propose our contribution towards enabling a widely-adopted shift in the methodology used to design analog circuits.

We present BAG, the Berkeley Analog Generator, an integrated framework for the development of generators of AMS circuits, i.e., parametric design procedures to synthesize a schematic and layout of a circuit according to a set of input specifications. We developed BAG with the goal of closing the gap between the designer and CAD communities. Designers can use BAG to develop circuit architectures closely following all steps of the custom flow. At the same time, BAG also assists them in defining an abstraction of the architecture, free of most implementation details (e.g. device sizing and technological parameters), in order to create a library of components suitable to be embedded within the desired optimization framework.

Using the previously introduced classification, BAG belongs to the knowledge-based category, in that the design flow is codified as a set of procedural scripts. We believe that this approach is more likely to be adopted by the designer community, because it maintains the central role of designers. Moreover, we argue that most optimization-based techniques proposed in the literature still require substantial design experience to produce high-quality results. At the same time, though, they enforce algorithmic steps in the design flow which can prevent the designer from fully driving the project towards the desired direction. We chose a dual approach. The basic version of the proposed flow is knowledge-based, so that the designer can maintain control. Specific sub-tasks can then be automated at the will of the designer to improve runtime and/or design performance. Instead of designing a specific circuit instance as in the standard custom flow, the designer uses BAG to develop a circuit generator, agnostic towards technology information and parameterized by the desired input specifications. The time overhead in setting up the flow is thus amortized by reusing the generator to synthesize circuit instances starting from multiple input specifications and across technology nodes. The availability of circuit generators also eases hierarchical top-down design, where complex blocks recursively instantiate sub-components fulfilling specifications propagated from the higher level.

In addition, we extend the state-of-the-art by developing BAG with the specific goal of supporting the designer in developing new generators. First, the framework is integrated, i.e., all steps of the architecture-design phase of the design flow, from block-level specification definition to correct layout implementation, can be performed in BAG. Second, we enriched BAG with helper classes, i.e., collections of template architectures and design routines – both at the schematic and at the layout level – to ease the development of circuit generators and relieve the designer from the burden of encoding commonly performed tasks. Within BAG, we designed generators for several circuits, including a Voltage Controlled Oscillator (VCO) and a Switched-Capacitor (SC) DC-DC converter. We show automatically-generated circuit instances in a commercial CMOS 65nm process, spanning a wide range of input specifications, both in terms of performance and physical footprint. Finally, we present preliminary results about the automatic migration of these designs to a commercial CMOS 40nm process.

We summarize our contributions as follow:

1) We introduce BAG, a design framework for AMS circuits capable of integrating all steps of the design flow, from architectural-specification definition to correct layout implementation, into procedural analog generators.
2) We developed helper classes to ease the codification of schematic sizing procedures and to simplify the creation of parameterized layouts for a rich set of layout styles.
3) We demonstrate the effectiveness of BAG by designing complex generators, capable of synthesizing multiple circuit implementations across different input specifications and two technology nodes.

The rest of the paper is organized as follows. In Section II, we introduce the BAG framework and provide an overview the proposed flow for the design of AMS generators. Section III presents the helper classes that aid the design codification process. Section IV gives examples of designed AMS generators. Finally, we conclude and discuss future research directions in Section V.

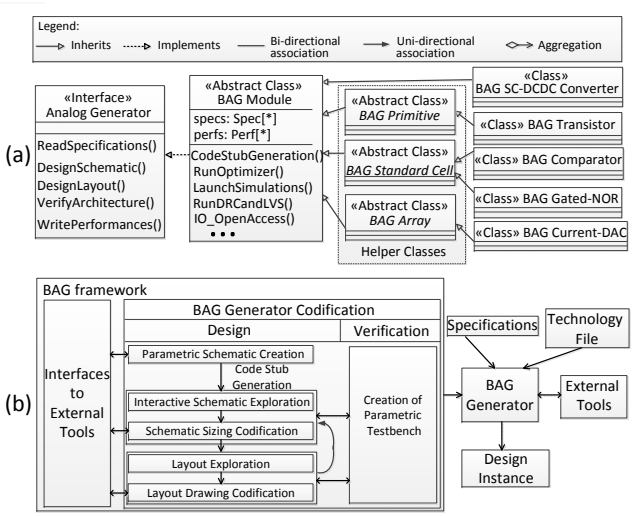


Fig. 1: A UML Diagram of the proposed generator interface with the BAG classes that implement it (a) and the BAG design flow (b).

II. Analog Generator Framework

In our approach to analog circuit automation a designer's deliverable is not a single instance of a sized schematic and clean layout for a particular circuit, but rather a generator for a desired class of circuits that can replicate, in an automated fashion, the design procedure that would have been used for a traditional, manual design.

A. Design Flow and Framework Structure

Any analog circuit generator should (at least) be capable of reading a set of input specifications, automatically sizing all schematic components, generating the corresponding physical layout, and outputting the resulting performances, which need to meet all specifications while optimizing some application-specific figure of merit. Borrowing terminology and graphical representation from the Unified Modeling Language (UML), we thus define the *interface* Analog Generator shown in Fig. 1(a). Each circuit generator is interpreted as a *class* which has to implement all the methods specified in the Analog Generator interface. Although individual generators, such as those in the rightmost column of Fig. 1(a), require specific procedures to optimize the performance of the design under analysis, much of the effort in creating a generator is not unique to any single circuit. BAG thus provides a set of *abstract base classes* that help a designer create new generators by providing interfaces to tools that perform common functions in the design process. Fig. 1(a) shows some of these functions as part of the main abstract base class, BAG Module, from which all analog generator classes inherit.

Ultimately, the process independence of a generator depends on the designer. Since it is difficult to design a technology independent generator without being able to debug or test it using a real technology, the recommended BAG design flow entails the selection of a representative process technology for use in the initial design exploration steps (described in more detail later in this section). By forcing the designer to access all technology-specific information through a well-defined interface, the job of translating the technology dependent design exploration into a technology independent generator is made more tractable, reducing the burden on the designer.

The BAG design flow, as shown in Fig. 1(b), begins in much the same way as an instance-based, manual design flow, i.e. by capturing a specific circuit architecture in schematic form. However, instead of entering a sized schematic, the designer creates a parametric schematic where only the connectivity among circuit devices is fully specified, while neither device sizes nor process information are provided. The purpose of the parametric schematic is to annotate as much of the designer's intent as possible. In order to do so, we created technology-agnostic primitive devices (e.g. NMOS and PMOS transistors, resistors, capacitors, etc.), whose sizes can be left blank (to be filled in later) or assigned meaningful parameter names to express, e.g., matching and ratio constraints. Primitives can also be assigned specific intent, e.g. a transistor can be annotated as *fast*.

Next, the designer creates a parametric testbench schematic in the same manner used for the design schematic, and enters the associated simulation setup – including simulation type, simulation parameters, probe points – through Cadence ADE. The parameterized design and testbench schematics are imported recursively in BAG and used to create stub class definitions for every unique cell in the hierarchy.

The designer can now – without writing any code – create an instance of the design. In the traditional flow, the designer debugs and explores the design by choosing some initial sizing, simulating, viewing the results, and iteratively adjusting the sizing until specifications are met. This flow can be replicated in BAG, with the important difference that the exploration is done in an interactive, code-based environment. Performing the initial exploration using the same code constructs that will be used in the final generator is key in lowering the designer effort in writing the generator code. For example, snippets of code used in the exploration process can be used to build the *DesignSchematic()* and *VerifyArchitecture()* methods required to implement the Analog Generator interface in Fig. 1(a).

The next step is creating a parameterized layout. Similar to the exploration step for schematic sizing, the designer can view the layout in an interactive environment where they can test changes before incorporating them into the *DesignLayout()* code. Once an initial layout has been created, the designer can return to the interactive environment and run physical verification checks – Design Rule Checking (DRC) and Layout Versus Schematic (LVS) – to be then added to the *VerifyArchitecture()* method. The designer can modify the layout and iterate until the design is DRC and LVS clean. The layout parasitics can then be extracted and post-layout simulations run. Using the results of these simulations, the designer can revisit the *DesignSchematic()* function and modify it as necessary to account for layout effects.

After the initial pass through the design flow, the designer can iteratively refine the generator class to improve the design performance, e.g. by calling numerical or equation based optimizers, and make the generator faster, more robust, and more reusable. In a complete generator, all knowledge of the design should be codified in the generator class definition. Once the generator is complete, the designer can pass input specifications and technology information to it in order to produce unique design instances of an architecture as depicted on the right side of Fig. 1(b).

B. Framework Development

Several platforms were considered for the implementation of the BAG framework, including Matlab, Perl, Python, SKILL and Tcl. Ultimately, Python was chosen for its strong support of object-oriented features such as abstract base classes and multiple inheritance. More expressive generators can be written by leveraging the plethora of off-the-shelf packages provided in Python, ranging from scientific computing (NumPy and SciPy), to graphical plotting (Matplotlib), and numerical optimization (e.g., CVXPY and GLPK). Python also has a powerful interactive command line that enables generator designers to rapidly iterate during the generator codification process.

The choice of the Python platform is further supported by the availability of the Synopsys' PyCell framework, which provides a (freely-available) API for creating process-independent parameterized cells (PCells) using "DRC correct-by-construction" functions [11]. At design time, these functions can be used to concisely express the desired relative placements of geometric objects. The enforcement of DRC correctness is instead postponed to compile time when the PyCell API reads the design rules from the technology file and performs the actual placement. For example, the designer can specify to "place two transistors as close as possible to one another" and a (possibly) different actual physical distance is automatically enforced for PyCell instances in different technologies. By building up hierarchical geometries using these functions, it is possible to create PCells that can be compiled for different technology nodes.

C. BAG Design Flow Example: Power Gated NOR

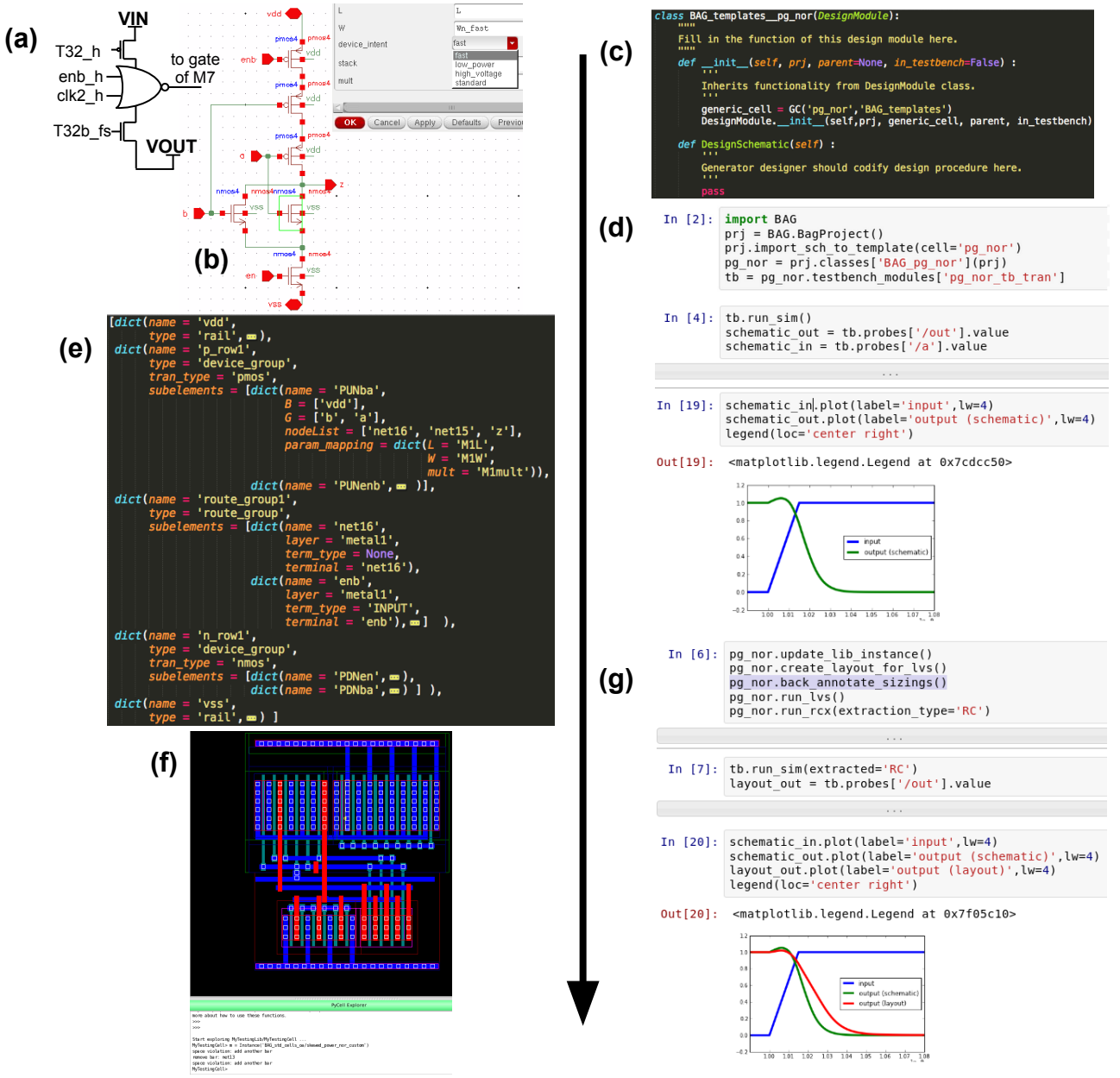
In this section, the design flow of Fig. 1(b) is elucidated through an example design of the power-gated NOR driver of Fig. 2(a). This design is relevant because it is used as a building block of the DC-DC converter presented in Section IV. Fig. 2(b) shows the entry of the parameterized schematic. One of the inputs of the nor gate is a high-speed signal, used to drive one of the switches of the converter, while the other input is a static enable signal. The designer captures this information by setting the width parameters of the two input NMOS transistors to the skewed values *Wn* and *Wn_fast*. Moreover, the designer can capture the desired *intent* for each device in the circuit, e.g. they can specify that the high-speed transistor be mapped to the fastest transistor available (e.g. a low threshold voltage (LVT) device) and the static device to a low-leakage device (e.g. a high threshold (HVT) device).

Next, the designer uses BAG to import the parametric schematic information and produce the stub class shown if Fig. 2(c) as well as a stub class for a transient testbench (not shown). These classes enable the designer to instantiate a power-gated NOR object and its associated testbench in the interactive Python environment shown in Fig. 2(d). From this environment, the designer can access the design hierarchy and parameters using a simple dot syntax, e.g. pg_nor.NM1.W or pg_nor.Wn_fast. The designer can then – within the interactive environment – explore the design by choosing an initial sizing, simulating, viewing the results, and adjusting the schematic sizes based on the results.

For custom digital cells like this skewed, power-gated NOR, the PyCell creation step can be fully automated. An intermediate data structure, readable by the Standard Cell helper class described in Section III-C, is automatically created by BAG (Fig. 2(e)) based on the circuit schematic. Fig. 2(f) shows a PyCell debugging environment provided by Synopsys where the designer can test further changes if needed before incorporating them into the PyCell code. Once an initial PyCell is compiled, the designer can return to the interactive Python environment to run physical verification checks (DRC and LVS), extract layout parasitics and rerun the testbench simulations (Fig. 2(g)). The designer now has enough information to implement the methods of the generator interface into the BAG Gated-NOR generator class of Fig. 2(a).

III. Helper Classes

Given a particular circuit topology, a designer typically makes assumptions about which design procedure should be used to size the circuit and how the layout should be structured. In many cases, these assumptions are shared across a specific category of circuit, so it is useful to create helper functions that aid with the specific sizing and layout procedures. We grouped them into helper classes associated with specific circuit categories in order to enable reuse and reduce the effort required for the initial codification of a generator. For layout related functions, we refer to these helper classes as layout *styles* which are implemented by abstract PyCell classes that designers can inherit from when designing PyCells for a specific generator.




```
class BAG_templates__pg_nor(DesignModule):
    """
    Fill in the function of this design module here.
    """
    def __init__(self, prj, parent=None, in_testbench=False) :
        '''
        Inherits functionality from DesignModule class.
        '''
        generic_cell = GC('pg_nor','BAG_templates')
        DesignModule.__init__(self,prj, generic_cell, parent, in_testbench)

    def DesignSchematic(self) :
        '''
        Generator designer should codify design procedure here.
        '''
        pass
```

```
In [2]: import BAG
        prj = BAG.BagProject()
        prj.import_sch_to_template(cell='pg_nor')
        pg_nor = prj.classes['BAG_pg_nor'](prj)
        tb = pg_nor.testbench_modules['pg_nor_tb_tran']

In [4]: tb.run_sim()
        schematic_out = tb.probes['/out'].value
        schematic_in = tb.probes['/a'].value

In [19]: schematic_in.plot(label='input',lw=4)
         schematic_out.plot(label='output (schematic)',lw=4)
         legend(loc='center right')

Out[19]: <matplotlib.legend.Legend at 0x7cdcc50>
```

```
In [6]: pg_nor.update_lib_instance()
        pg_nor.create_layout_for_lvs()
        pg_nor.back_annotate_sizings()
        pg_nor.run_lvs()
        pg_nor.run_rcx(extraction_type='RC')

In [7]: tb.run_sim(extracted='RC')
        layout_out = tb.probes['/out'].value

In [20]: schematic_in.plot(label='input',lw=4)
         schematic_out.plot(label='output (schematic)',lw=4)
         layout_out.plot(label='output (layout)',lw=4)
         legend(loc='center right')

Out[20]: <matplotlib.legend.Legend at 0x7f05c10>
```

```
[dict(name = 'vdd',
      type = 'rail', ),
 dict(name = 'p_row1',
      type = 'device_group',
      tran_type = 'pmos',
      subelements = [dict(name = 'PUNba',
                          B = ['vdd'],
                          G = ['b', 'a'],
                          nodeList = ['net16', 'net15', 'z'],
                          param_mapping = dict(L = 'M1L',
                                               W = 'M1W',
                                               mult = 'M1mult')),
                     dict(name = 'PUNenb', )],
 dict(name = 'route_group1',
      type = 'route_group',
      subelements = [dict(name = 'net16',
                          layer = 'metal1',
                          term_type = None,
                          terminal = 'net16'),
                     dict(name = 'enb',
                          layer = 'metal1',
                          term_type = 'INPUT',
                          terminal = 'enb'), ]  ),
 dict(name = 'n_row1',
      type = 'device_group',
      tran_type = 'nmos',
      subelements = [dict(name = 'PDNen', ),
                     dict(name = 'PDNba', ) ] ),
 dict(name = 'vss',
      type = 'rail', ) ]
```

Fig. 2: Design flow of a power-gated NOR driver (a) illustrating parameterized schematic entry (b), class stub generation (c), interactive schematic exploration (d), fully automated PyCell generation (e), interactive layout exploration (f), and layout verification and post-layout simulation (g).

A. *Primitive Device*

BAG includes helper classes for primitive devices that enable device characterization for any technology node. This enables designers to write sizing procedures in terms of technology independent function calls that reference the characterization data lookup tables. These helper classes also enable the automatic mapping of device intent, based on the device characterization data, to specific devices in a given technology.

Many knowledge-based design techniques rely on accurate models of basic device characteristics for use in analytical calculations. For example, the g_m/I_d sizing methodology [12], commonly used to design analog circuits such as amplifiers, requires the ratio of the small-signal transconductance to drain current as a function of device sizing and bias voltages. The characterization helper class for transistors builds and maintains a lookup table containing the small-signal parameters for different bias conditions, channel lengths, channel widths, process corners, and temperatures. The class includes a *size_for()* function that allows the designer to set the device width based on the desired g_m/I_d.

The SC DC-DC regulator generator presented in Section IV-B uses the Capacitor Primitive helper class to characterize the capacitor density and bottom-plate ratio of the capacitor cell. These data are critical for sizing all components of the regulator. In the plots of Fig. 3, each point corresponds to a simulation – in both 65nm and 40nm processes – of a post-layout netlist extracted from a capacitor cell consisting of a MOS capacitor in parallel with a Metal-On-Metal (MOM) capacitor. The cell has a programmable size, aspect ratio, polysilicon density, and number of metal layers for the MOM

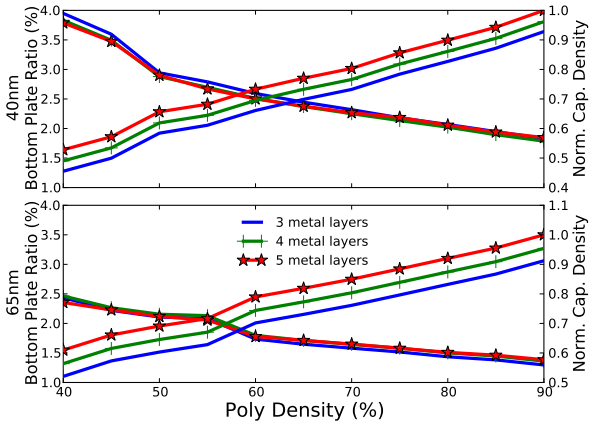


Fig. 3: Extracted capacitor density and bottom plate ratio simulation results for a unit capacitor PyCell in commercial CMOS 40nm and 65nm processes.

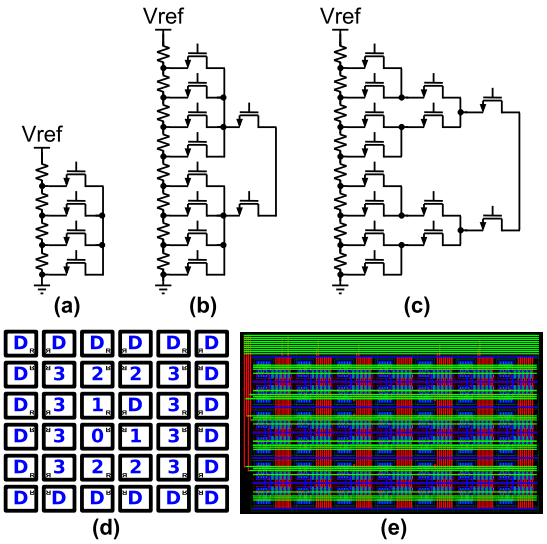


Fig. 4: Three specific instances of a variable structure architecture for a resistor string DAC with $bits = 2$, $radix = 4$ (a), $bits = 3$, $radix = 4$ (b), and $bits = 3$, $radix = 2$ (c). Floorplan of the array layout style (d) and a 4-bit instance of a bias current-DAC (e).

capacitor. Being able to characterize the capacitor cell allows the designer to make informed design tradeoffs during the initial stages of the design and reduce the number of iterations required to converge to a final sizing.

B. Variable Structure and Array

Although BAG generators typically have netlists with fixed structures, some types of circuits have netlists that vary in a predictable manner based on their input specifications. For example, the number of bits in a resistor string DAC specifies the dimensionality required for the array of resistor unit cells and a radix parameter also specifies the connectivity of the switch network used to select one of the voltages from the resistor string and connect it to the DAC output. Fig. 4(a)-(c) illustrate the array and tree structures for the resistor string DAC example across several combinations of bit and radix specifications. BAG supports common variable structure generators, including arrays

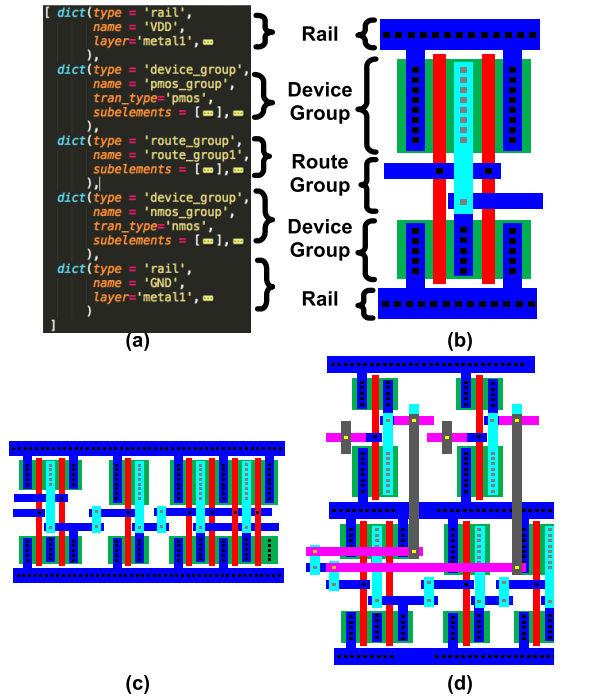


Fig. 5: Python data structure for Standard Cell style (a) and layouts examples for Standard Cell style (b), Standard Row style (c), and Standard Block style (d).

and trees, with a BAG Array class that can procedurally create the circuit schematic structure using the base subelements of the structure and a simple specification of the pattern.

BAG also includes an Array layout style to simplify the creation of PyCells with a floorplan similar to the one shown in Fig. 4(d) where an array of unit cells is connected using a common-centroid (or other) pattern and surrounded by dummy unit cells. The Array layout style allows the designer to parameterize the number of cells (or number of bits), the placement pattern, the symmetry of adjacent cells, and the number of rows of dummy cells. It also includes functions to help route signals to the edges of the array. Fig. 4(e) shows a 4-bit instance of a current-DAC PyCell which uses the Array layout style.

C. Standard Cells, Rows, and Blocks

The Standard family of layout styles is intended to help designers build hierarchical circuit layouts by defining a standard geometric interface that can be automatically stacked and abutted and includes a straightforward routing scheme. Although the Standard family of styles is useful for creating small blocks of custom digital cells, it is intended to handle many types of AMS circuits including amplifiers, comparators, decoupling and coupling capacitors, small DACs, level shifters, and non-overlap circuits. Unlike digital Standard Cells which have a fixed vertical pitch, BAG Standard style cells have programmable aspect ratios that make them ideally suited for placement around and between other AMS blocks with more rigid layout requirements.

The basis for the Standard family of layout styles is the Standard cell style. Standard Cell style is used to implement leaf cells, i.e. cells that are built entirely from primitive elements such as the simple amplifier of Fig. 6(a)). All of the Standard styles are implemented as abstract PyCell classes that read a data structure entered by the designer and translate it into a layout. The data structure defining Standard Cells, shown in Fig. 5(a)), consists of Python dictionaries that provide details about the layers and relative locations of power

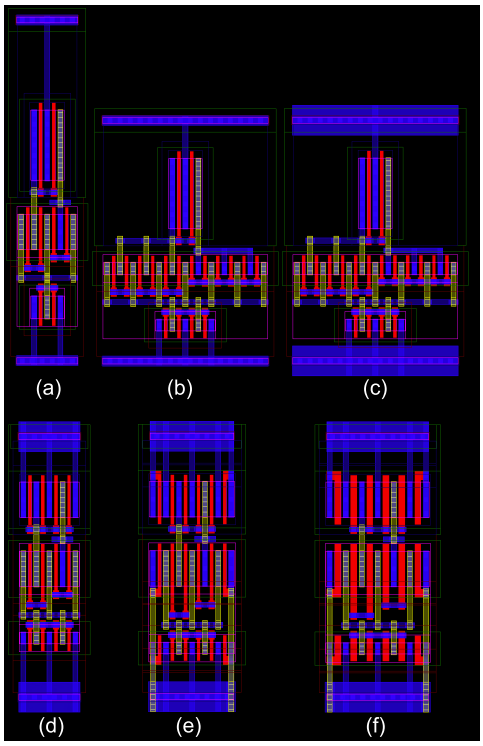


Fig. 6: An amplifier PyCell (a) is modified with reduced vertical pitch (b), increased power rail width (c) equalized NMOS and PMOS regions (d) gate dummies (e) and increased channel length (f).

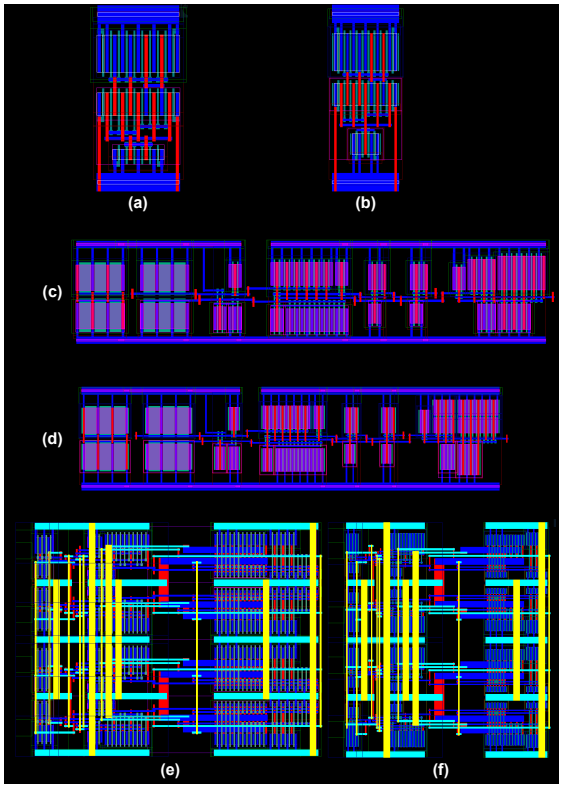


Fig. 7: Layouts in commercial 65nm and 40nm processes for a Standard Cell style amplifier (a) and (b), a Standard Row style asynchronous comparator (c) and (d), and a flying inverter switch driver (e) and(f).

rails, transistor devices, and cell terminals and internal routing nets. For many cells, creating this data structure is sufficient to produce a DRC and LVS clean layout, however, for more complex cells, the designer can extend or override the functionality of the Standard Cell class. For custom digital cells, the data structure is automatically generated as shown in Fig. 2(e). Standard Cell PyCells have several parameters that allow the designer to control the vertical pitch of the cell, the width of power rails, the vertical space dedicated to NMOS vs. PMOS devices, whether the cell is in deep nwell or not, the flavors of the devices (LVT, HVT, thick oxide...), whether (and where) to add dummies to each device, whether to add vertical pins on either side of the cell, and several more. Fig. 6 illustrates several of these capabilities for an amplifier circuit. Another parameter allows the class to adjust the ratio of vertical space dedicated to nmos vs. pmos automatically in order to equalize the number of pmos fingers to the number of nmos fingers and avoid wasting space. This is illustrated by the extra space in Fig. 6(c) which is gone in Fig. 6(d).

The power of the Standard family of layout styles is fully realized when several Standard Cells are connected together and the interconnected block still retains a flexible aspect ratio. BAG provides two styles to allow the designer to connect several Standard Cells. The Standard Row style (Fig. 5(c)) is used to connect a row of Standard Cells if their inputs and outputs are connected only to adjacent cells, while the Standard Block style (Fig. 5(d)) has the capability to stack rows of Standard Cells (or Standard Row objects) and includes a more complete inter-cell routing mechanism. Because the Standard family of layout styles are written using the "DRC correct-by-construction" functions from the PyCell API, Standard style PyCells can be compiled for different technology nodes. Fig. 7 shows three different Standard style PyCells that were all compiled and verified as DRC clean in commercial processes at the 65nm and 40nm technology nodes. The Standard Cell style amplifier of Fig. 7(a)-(b) was compiled with the same device width and cell pitch for both nodes as were the Standard Row style asynchronous comparator of Fig. 7(c)-(d) and the Standard Block style flying inverter of 7(e)-(f).

IV. Case Studies

In parallel with the development of BAG, several designs have been carried through the BAG design flow. Building on the layout style helper classes presented in Section III-B and Section III-C, we first present a custom PyCell of an important functional block (voltage-controlled oscillator) that does not map directly to a layout style but can be decomposed into blocks that do. Next, we present a complete generator example of a full system (DC-DC converter).

A. Voltage Controlled Oscillator

Voltage Controlled Oscillators (VCO) are an essential building block in phased-lock loops (PLL) and frequency synthesizers. Of the two main varieties of VCOs, ring and LC, LC-based VCOs are typically characterized by lower phase noise and jitter at equal power consumption so this topology is often used for communication circuits and other applications where low noise is critical. Depending on the target application (e.g. GSM, Wi-Fi, ZigBee), different trade-offs are made among the oscillator's most sensitive performance specifications, i.e., phase noise, power consumption, and frequency tuning range. It is thus appealing to codify the design flow of an LC-VCO in BAG to reduce design time and ease the system-level exploration of multiple different implementations.

The chosen LC-VCO topology is depicted in Fig.8(a). It consists of a cross-coupled transistor pair, a current source, and an inductive-capacitive (LC) tank load. The product of the LC values in the tank sets the oscillation frequency of the VCO. Since it is often useful to dynamically tune the oscillation frequency (e.g. to select

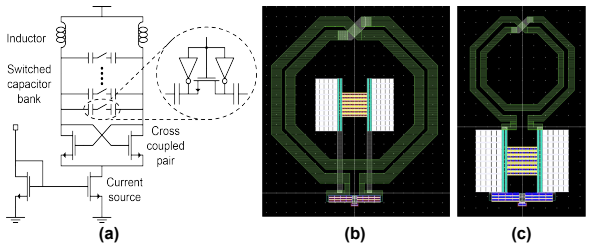


Fig. 8: LC VCO schematic (a) and layout instances for VCOs with large (b) and small (c) diameter inductors.

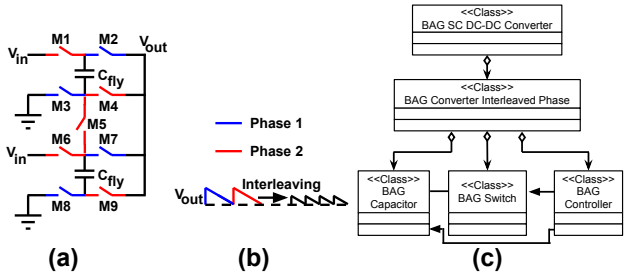


Fig. 9: SC DC-DC converter topology (a), operation (b), and UML diagram (c).

the desired channel in a communication link), a Switched-Capacitor (SC) bank has been added to the tank to modify the capacitance value. The implemented LC-VCO PyCell inherits from the Array (Standard Cell) helper class a template architecture to implement the SC bank (transistor pair and current source). It has multiple input parameters that can be adjusted to achieve different performance specifications. For example, the center frequency of the tank can be set by adjusting the inductance value, which is a function of the physical diameter and number of turns of the inductor; the desired dynamic range and resolution in tuning the oscillation frequency can be achieved by designing an SC bank with a suitable number of bits. Finally, in the case of large inductor size, the PyCell automatically places the SC bank underneath the inductor in order to save area, as shown in Fig.8(b).

B. Switched Capacitor DC-DC Regulator

The most complex circuit generator created so far is a fully-integrated Switched-Capacitor (SC) DC-DC regulator, which extends the work of [13] by equipping the converter with a controller capable of maintaining a fixed output voltage in the presence of steps in the load current. As the number of digital cores and peripheral blocks with separate voltage supplies in modern SoCs increases, integrated switched voltage regulators are becoming increasingly relevant because they are capable of providing (multiple) voltage supplies with high power-conversion efficiency without requiring costly external components. In order to ease the floor-planning of these SoCs it is then useful to adapt the regulator aspect ratio in order to fit the empty space available on die.

SC converters, like the step-down converter in Fig. 9(a), typically switch a flying capacitor in two phases to deliver power to the digital load while converting the voltage level. Dividing the capacitor into several separate capacitors and interleaving their control signals can help reduce the voltage ripple inherent to SC operation. The number of interleaved phases, topology of the SC converter, frequency of switching, size of the flying capacitors, and size of the switches determine the exact relationship of the output voltage to the input voltage and the efficiency of the power conversion. Further, a feedback loop can be used to dynamically adjust the switching frequency and lock the output voltage to a reference. Fig. 10 shows a top-level schematic simulation – run through BAG – of the full regulator with a regulating control loop responding to negative (left) and positive (right) load current steps.

The top level UML diagram of the DC-DC regulator generator is shown in Fig. 9(c). A regulator instance is made of one or more interleaved phases (the exact number is determined at optimization time), each made of power switches, capacitors and control logic. The hierarchical structure of the diagram can be ported directly to the BAG framework. We created generators for each sub-block to be called multiple times within the higher-level generators to instantiate *objects*, i.e., sized and laid-out instances, of the sub-blocks. We emphasize that the specifications of each sub-block are set hierarchically within higher level generators in order to optimize global performances, e.g. to create as compact of a layout as possible.

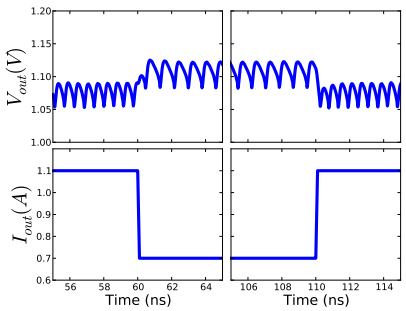


Fig. 10: SC DC-DC Regulator simulated transient response.

Analog designers often begin their system-level exploration by writing simple behavioral models and equations to capture the first order behavior of the system. Based on the analytical optimization flow presented in [13], we created a Python function that accepts as inputs basic process characterization data (obtained by reading the technology database provided by the Primitive Device helper classes) switch impedance, capacitance density (see Fig. 3), and target output power specifications, and it determines as outputs: 1) the optimal switch and capacitor sizing for the DC-DC regulator core and the optimal number of interleaved phases, at the schematic level, and; 2) the physical dimensions of the flying capacitors in order to obtain the desired top-level aspect ratio, at the layout level.

The outputs of the system-level optimization are hierarchically passed to the circuit generators for all the sub-blocks. Fig. 11 shows the hierarchical layout of the regulator. The top-level DC-DC regulator generator inherits from the Array helper class (Section III-B) to instantiate the desired number of interleaved phases (Fig. 11(a)). Each interleaved phase is made of flying capacitors, switches and their drivers, and logic to generate the control signals for the switch drivers (Fig. 11(c)). The generator of the control logic, i.e., level-shifters and non-overlap circuitry, inherits from the Standard Block helper class (Section III-C). The gates in this block get sized using a script which aims to minimize power consumption while meeting a constraint in the length of the critical path. Finally, switches and drivers are laid-out in two rows (Fig. 11(d)) mimicking the schematic representation (Fig. 9(a)). The drivers inherit from the Standard Cell helper class, and we can use their capability of adapting the pitch to minimize the white space in this block. We fix the width of the bottom row, which contains the most complex drivers [13], to meet electro-migration specifications. We then run a binary search on the pitch of the top row (Fig. 12) to match the width of the bottom one. This is an example of how encoding the designer knowledge can quickly produce very compact layouts that would have been hard to obtain using an agnostic constraint-based layout generator. Finally, in Fig. 11(e), we show the layout of the switch/driver block migrated to the 40nm process, the largest cell that we have been able to migrate so far.

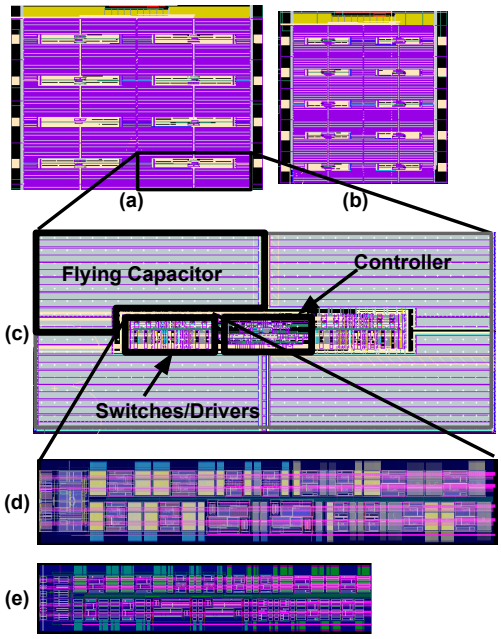


Fig. 11: Layout for a high power density, high output (a) and a low power density, low output power (b) DC-DC regulators. More details of the hierarchical structure of the regulator (c-d) and switch/driver block layout migrated into a commercial CMOS 40nm process (e).

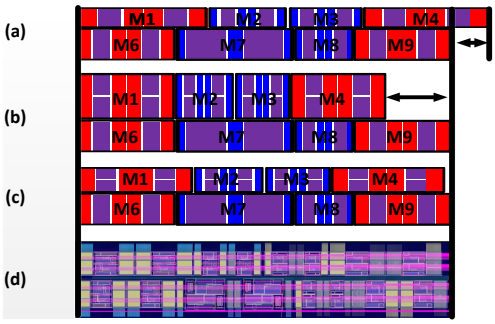


Fig. 12: Binary search (a)-(c) and resulting layout (d) of a the switch/driver block of the DC-DC regulator

After the first iteration of the design flow has been completed, pre and post-layout simulations are run to evaluate the regulator performance, and the sizing flow is iterated to take the updated parasitic information into account. We emphasize that the automation of the design flow substantially lowers the cost of each iteration (in our implementation, each iteration is fully automated, and it takes around three hours to run), thus enabling a tighter tuning and verification of the regulator performance with respect to a manual flow.

In order to fully validate the functionality and flexibility of the BAG framework and to verify the quality of the synthesizable circuits, we will evaluate 3 different implementations of the SC DC-DC regulator each targeting different specifications in terms of total output power (ranging from $100mW$ to $1W$), power density ($0.2W/mm^2 - 1W/mm^2$) and aspect ratio ($0.75 - 1.15$). The layout of two of the regulators are shown in Fig. 11(a-b).

V. Conclusions

We argued the need for a paradigm shift in AMS circuit design, which should no longer be focused on producing a sized schematic and layout of a singular circuit instance, but circuit generators, i.e. parametric design procedures to synthesize schematic and layout of the circuit according to a set of input specifications. We defined the requirements to create such analog generators and implemented a Python-code framework called BAG to ease their creation. BAG enables analog circuit designers to create analog generators through the direct codification of the manual circuit design process. Numerous generators have been written within the framework including a switched-capacitor DC-DC regulator – a complex hierarchical system relevant to modern SoC architectures.

As future work, we plan to populate a library of generators to demonstrate the efficacy of BAG. In particular, three different SC DC-DC regulator instances, targeting different specifications, will be fabricated in a commercial CMOS 65nm and we will migrate the full design to CMOS 40nm to explore how to write generators to enable full technology independence. We will also release the framework and several example generators under an open source license [14].

VI. Acknowledgment

This research was supported by the UC Discovery Grant ele07-10283 under the IMPACT program, NSF Infrastructure Grant No. 0403427, and partially funded by DARPA Award Number HR0011-12-2-0016. The authors acknowledge the sponsors, faculty, students, and staff of the Berkeley Wireless Research Center for their support and Jaeha Kim for useful discussions and sharing code.

References

[1] ITRS. (2011) International technology roadmap for semiconductors. [Online]. Available: http://www.itrs.net/Links/2011ITRS/Home2011.htm

[2] G. G. E. Gielen and R. Rutenbar, "Computer-Aided Design of Analog and Mixed-Signal Integrated Circuits," *Proceedings of the IEEE*, vol. 88, no. 12, pp. 1825–1854, 2000.

[3] M. G. R. Degrauwe *et al.*, "IDAC: an interactive design tool for analog CMOS circuits," *IEEE Journal of Solid-state Circuits*, vol. 22, pp. 1106–1116, 1987.

[4] R. Harjani, R. Rutenbar, and L. Carley, "OASYS: a Framework for Analog Circuit Synthesis," *IEEE Transactions on CAD*, vol. 8, no. 12, pp. 1247–1266, Dec. 1989.

[5] U. Choudhury and A. Sangiovanni-Vincentelli, "Automatic Generation of Parasitic Constraints for Performance-Constrained Physical Design of Analog Circuits," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 12, no. 2, pp. 208–224, Feb. 1993.

[6] G. Van der Plas *et al.*, "AMGIE-A Synthesis Environment for CMOS Analog Integrated Circuits," *IEEE Transactions on CAD*, vol. 20, no. 9, pp. 1037–1058, Sep. 2001.

[7] R. Rutenbar, G. Gielen, and J. Roychowdhury, "Hierarchical Modeling, Optimization, and Synthesis for System-Level Analog and RF Designs," *Proceedings of the IEEE*, vol. 95, no. 3, pp. 640–669, Mar. 2007.

[8] M. Horowitz *et al.*, "Fortifying Analog Models with Equivalence Checking and Coverage Analysis," in *Design Automation Conference (DAC), 2010 47th ACM/IEEE*, 2010, pp. 425–430.

[9] P. Nuzzo *et al.*, "Methodology for the Design of Analog Integrated Interfaces Using Contracts," *Sensors Journal, IEEE*, vol. 12, no. 12, pp. 3329–3345, 2012.

[10] I. Cadence Design Systems. (2012) Virtuoso Analog Design Environment GXL. [Online]. Available: http://www.cadence.com/rl/Resources/datasheets/virtuoso_adeGXL_ds.pdf

[11] Synopsys. (2012) Pycell Studio. [Online]. Available: http://www.synopsys.com/Tools/Implementation/CustomImplementation/Pages/PyCellStudio.aspx

[12] 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," *Solid-State Circuits, IEEE Journal of*, vol. 31, no. 9, pp. 1314–1319, 1996.

[13] H.-P. Le, S. Sanders, and E. Alon, "Design Techniques for Fully Integrated Switched-Capacitor DC-DC Converters," *Solid-State Circuits, IEEE Journal of*, vol. 46, no. 9, pp. 2120–2131, Sep. 2011.

[14] "BAG framework." [Online]. Available: http://www.eecs.berkeley.edu/~crossley/BAG