Generators, Templates, and Code Generation for Flexible Automation of Array-Style Layouts

Benjamin Prautsch[*], Reimund Wittmann[†], Uwe Eichler[*], Uwe Hatnik[*], Jens Lienig[§]
[*]Fraunhofer IIS/EAS, Institute for Integrated Circuits, Division Engineering of Adaptive Systems, Dresden, Germany
{Benjamin.Prautsch, Uwe.Eichler, Uwe.Hatnik}@eas.iis.fraunhofer.de
[†]IMST GmbH, Kamp-Lintfort, Germany, reimund.wittmann@imst.de
[§]Dresden University of Technology, Dresden, Germany; jens@ieee.org

***Abstract*—The design of integrated circuits from the specification onward aims at the successful validation by silicon measurements. One key milestone in this process is the completion of the layout. This, however, is very challenging as many iterations are usually necessary due to parasitic effects. In order to address this challenge in analog layout design, our work extends procedural generator-based automation. A declarative array template is embedded into the common generator structure. Following this structure, generator code is automatically generated with a schematic as the input. Using this approach, a flexible generator is created immediately that allows automatic design of array-style layouts with template-based flexibility and at generator-based execution speed. In addition, the template enables early and fast parasitic estimates. Our combined approach contributes to analog layout automation by bridging the gap between generators and templates.**

Keywords—Analog automation, generator, template, array layout, reuse, design migration, analog layout

I. Introduction

Analog integrated circuit design is a very sophisticated task that requires much expertise and consumes considerable efforts. Contrary to digital design, analog suffers from a lack of automation which results in long design time and limited reuse. As technologies evolve and become more advanced while cycle times are getting ever shorter, analog design engineers must meet increasingly challenging schedules. Thus, several approaches are followed to accelerate design.

Most pragmatically, design engineers concentrate on few types of designs so that they become very familiar with them. This increases productivity as they can then take known best-practice decisions (e.g., hierarchical organization or additional unconnected layout devices). However, such steps are all manually done. Thus, in the last decades several additional features were included into common design environments in order to improve productivity. Well-known examples are schematic-driven layout (SDL), placement tools for matching structures, visually-assisted automation, or instant design rule check (DRC). These environments are of great help, however, still most parts of analog are implemented manually.

Lately, tools that support more automation through reuse and generation of cells begin to find their ways into design environments. For layout reuse, flexible templates are used [1, 2]. They usually represent the layout arrangement in an abstract way and guide layout creation through optimization approaches [3, 4, 5]. Subsequently, the actual layout generation step follows which, depending on the capability of the generation engine, might support multiple process technologies. At this level, procedural layout *generators* come into picture [6, 7, 8, 9, 10] that automate generation of devices and building blocks. The advantage of generators is that they provide fast and parameterizable solutions to known problems employing expert knowledge [11]. However, their drawback is that they often lack flexibility and hardly allow structural changes as adapting their source code often requires significant development time. *Templates*, on the other hand, allow much flexibility in layout definition; they also help considering (geometric) constraints that are essential for analog layout automation [12, 13, 14].

We believe that the combination of templates and generators is very promising to tackle the aforementioned shortcomings. In [15], generators are combined with optimization based on swarm intelligence showcasing the advantage of combining flexible techniques with generator-based approaches. More generally, [11] proposes a *bottom-up meets top-down design flow* approach besides a continuous layout design flow in order to tackle analog layout automation.

In this work, we combine the bottom-up generator approach with the top-down template approach and add automatic schematic-to-code creation, which, to the best of our knowledge, has not been done before. This way, generators are automatically derived from an input schematic while incorporating the known flexibility of a template for user-defined placement. As the result, generator-based automation with more flexibility and faster generator development becomes possible. Our approach combines the following contributions into a single flow (Fig. 1):

- Generator framework that is technology-agnostic [16],
- Template approach for flexible placement and routing,
- Automated source code creation for fast generator availability (that can also be extended manually).

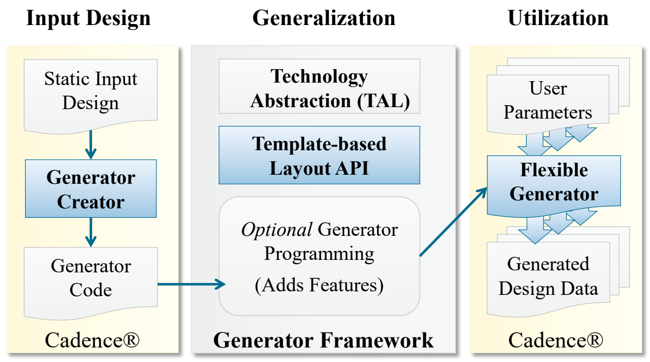

Figure 1 Illustration of the flow for automated generator code creation and utilization (with our contributions highlighted in blue). First, a static schematic is analyzed by the Generator Creator. All PDK devices are then mapped through technology abstraction (TAL [16]) and assigned to initial positions of the array template in the generator code. The resulting generator can be executed by the user. It allows to (re)arrange and adapt the layout generated for appropriate results across sizing and PDKs. Optionally, the generator code can be extended in order to implement additional functionality.

II. Our Design Flow Approach

When utilizing generators, the analog design flow is often extended by an additional hierarchy of (generated) building blocks. Instead of designing static blocks, procedural generators describe and create the required views flexibly. As designers might be required to programing code, it has even been called a "paradigm shift in AMS circuit design" [17]. So far, however, generator programming has been the bottleneck.

A. *Automated Generator Creation*

As designers are usually not into programming and as every design differs, automated generator code creation is desirable. Therefore, we follow an automated approach that creates flexible generators for schematic, symbol, and layout automatically based on a static input design. Our approach standardizes the code structure, thus, simplifies generator development and code creation. At the same time, valuable expert knowledge (being a key aspect of generators [11]) is stored in an executable way in order to accelerate both design and reuse.

In Fig. 1, the automatic process of generator creation is shown. Using our *Generator Creator*, an existing schematic (and symbol) is converted into parameterizable generator code. It can exactly replicate the input schematic through a default parameter set, creates a similar symbol (or a standard symbol), and assigns the instances found in the schematic to the abstract array-style template (see Section II.B). Moreover, a technology abstraction layer (TAL) [16] is used in order to map particular technology-dependent parameters to a generic representation (e.g. adaptation of the name of the width parameter of a transistor or the type of a transistor). The generator code created is the basis to include additional functionality into the generator. Programming the initial code manually would consume significant development time.

Once the generator is created, it can be parameterized and executed in order to create variants of building blocks of a design including the related array-style layouts. As the result, the created generator can be used for designs with different specifications or process technologies. This, in turn, enables fast layout prototyping and design.

B. *Template-based Layout*

Templates are widely used in optimization-based approaches in order to define floorplanning and partly placement and/or routing. They are abstract definitions of the target layout. In contrast to generators, they are not executable. Examples are LDS [1], dynamically generated templates in AIDA [18], as well as methods that use templates for porting like IPRAIL [4] or the fast prototyping approach in [5]. Templates have different appearances. They can be described in a flexible way [1, 19] or represent regular structures, such as arrays [20] or "streets" [21]. Some EDA approaches also use templates for both placement and routing [21, 22].

Inspired by the template approach, this work follows the idea of structuring procedural generator code using flexible layout templates for the layout description (currently limited to array-styles). This is beneficial as programming structural/topological variants, flexible sizing, placement options, and routing at the same time is hard to maintain in procedural code. Instead, our approach first describes the abstract layout arrangement declaratively using a template (that has to be implemented before). In the next step, this description is analyzed and translated into positions for each element of the template.

Based on a given placement, the following routing strategy is applied to the array. The regions between instance columns form the routing channels. Their size is not static but an adaptive result from the routing contained (implemented as composite pattern). The routing approach assumes unit devices but is not limited to it and connects two nets each. The method first prioritizes cohesive unit devices available on either side of a routing channel. The algorithm prevents occupying neighboring routing channels (which means that only two in three routing channels will be used to connect similar devices). Then, the remaining devices are collected and assigned to the routing channels in the same way while starting with the least filled ones. Subsequently, vertical wires are drawn to each unit device (from bottom to the most distant logical device) and a horizontal trunks plus vias follow. Finally, a procedural command sequence translates all these abstract elements defined in the template to "real" instances, wires, and vias in the actual layout.

As a result, layout description and program flow are structured separately which improves flexibility and eases generator development. Utilization of the *Generator Creator* also reduces faults significantly as the code is generated automatically and as checks for logical errors are included using callbacks (see III.B). This means that it is possible to promptly detect fatal LVS errors such as missing instances in case of wrong user inputs or faulty (manual) code updates. A valid LVS finalizes the verification of a generated building block.

As the template controls the first part of the layout sequence (see III.B), it can afterwards be combined with procedural code. This way, much effort for placement can be automated in order to reduce overall code development time of custom generators. Additional generator code can, among others, include details of the layout generation such as additional shapes, vias, well contacts, or routing details.

III. Application in Our Generator Framework

This work extends the functionality of our generator framework in [9]. The *Layout API* (application programming interface) is included into the generator structure adding array-style template flexibility. Our *Generator Creator* has been adapted in order to automatically create generator code that controls this template via the generator GUI.

A. *Layout API*

The *Layout API* is an extension to the generator API that adds template capability. It allows to use declarative and procedural code within the same generator. In this work, it uses (among others) a grid-based approach comparable to the approach in [20], meaning the layout area is segmented into rows and columns. Each row's height can vary as well as each column's width. Fields of the grid can be defined to represent a device or a hierarchical template recursively. The generator that utilizes this API requires only few code lines to define and interpret a template in order to create the actual layout at the level of the design environment.

B. *Programming Interface and Generator Structure*

Generators implemented with our method apply a similar code structure using class inheritance. Particular tasks are separated in specialized methods which are then executed by the base class for each generator (Fig. 2). First, the code defines user parameters which are shown in a parameter mask (for interaction with the designer) or hierarchically passed through the generator hierarchy. Second, callbacks are defined that automatically consider cross-dependencies between parameters in order to validate user inputs. Third, common properties of all views of the generated cell(s) are defined in the *prepare()* method in order to ensure consistency. Finally, respective methods are implemented to describe the circuit representations (views) for schematic, layout, and symbol.

The template has been included into this structure in order to enable user configuration (*param_spec()*), run checks (*param_check()*), and control the layout process *layout()*.

```
class Generator(gen.HierBlock):
  def param_spec(self): # parameters and constraints
    self.template = Template(…)
  def param_check(self):# parameter updates & callbacks
    self.template.update(…)
  def prepare(self):    # common data for all views
    self.template.evaluate(…)
  def schematic(self, cv):
    # procedural schematic description
  def layout(self, cv):
    self.template.draw(cv) # template execution
    myShape = cv.create_shape(…) # optional code
```

Figure 2 Basic structure of (hierarchical) generator code. The names of the class methods are predefined in the abstract base class that implements the general execution order of all methods when run. In addition to schematic and layout, also methods for symbol and testbench creation can be defined in the generator code.

C. *Design Reuse with the Generator Creator*

Our Generator Framework [9] interfaces with the Cadence® Virtuoso® design environment where the *Generator Creator* can be run in order to translate existing schematics into generator code. The *Generator Creator* executes the following steps: (1) the schematic is read and all elements of it are stored, (2) all data fetched is converted into a PDK-independent description, (3) the generic data is translated into generator code fragments that represent the generator structure given in Section III.B., and (4) the fragments are merged to a complete generator file. This method tremendously accelerates generator development as thousands of lines of code are created within seconds.

Each created generator can be run immediately or whenever required. As the generator code links parameter mask and *Layout API*, entry fields allow to modify the template. As default, an (almost) quadratic array is defined with the devices assigned. Based on user input, the arrangement can be adapted (e.g. number of matrix rows or the placement algorithm). All parameters entered into the generator GUI can be saved, too, in order to reuse them across projects and even PDKs along with the generator. For example, the placement definition can be reused this way.

IV. Design Experiment

We validated our presented approach on a capacitor arrangement that has a regular structure based on unit devices. It is part of a SAR ADC architecture [23, 24] that we intend to build for very complex matching constraints. Such structures cannot efficiently be designed manually by following best practices (see Section I). This prevents *close to optimal* unit capacitor placement with minimal influence of the interconnects. The main unit capacitor array of the popular differential split-cap architecture consists of 2 sub arrays, each again split into two arrays for MSB and LSB (most and least significant bit) values. So, in total, 4 sub DACs have to be combined into a single layout for high sub DAC linearity, high MSB accuracy, N channel and P channel symmetry, and matching requirements. The capacitor ratios have to match in presence of inherent layout parasitics in order to meet the SAR target resolution requirements. It is not easy to find a well-working layout arrangement, as the unit capacitor size and total capacitor array configuration strongly influence both random and systematic errors of the circuit.

Using our combined template and generator approach, we are able to fast try placement options and estimate the related parasitic effects. While manual placement would be very time-consuming, the *Generator Creator* combined with the template approach results in a flexible generator to allow fast early analyses. An example schematic with 1024 capacitors (twice 256 for LSB and 252 for MSB plus eight coupling capacitors) was defined. They represent the capacitances of the block diagram shown in Fig. 3. In order to achieve a well-matched placement, the logical devices must be split into unit devices which are arranged close and with sufficient size [25]. At the same time, a common-centroid pattern compensates process gradients [26]. In order to achieve a good arrangement, we adapted the common-centroid algorithm from [27] which creates a highly dispersed placement.

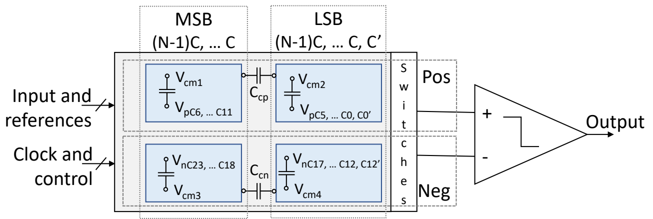

Figure 3 Coarse block diagram of the SAR ADC core. The capacitor array instantiates the capacitors C0 to C23 (and dummies C0' and C12') for MSB and LSB on both positive (Pos) and negative (Neg) channel. The split cap coupling capacitors are indicated, too.

Additionally, systematic errors from the routing are to be considered, too, as even a perfectly matched placement will be degraded significantly by parasitic routing capacitances [28], especially with very small capacitances [29].

The overall time from schematic import to initial layout generation is only a few minutes, with an overall layout generation run time of about 70 seconds. A reusable generator is created, too. This way, large parts of the layout can be generated much faster than manually possible, which enables optimization over parameter variants and arrangements. A generated capacitor arrangement is shown in Fig. 4.

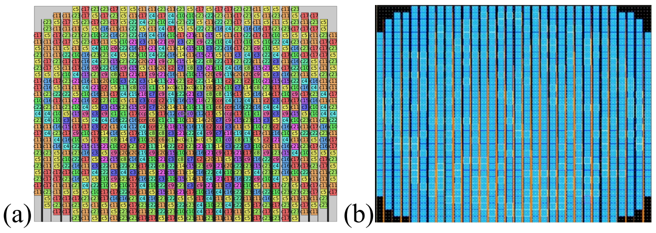

Figure 4 Generated capacitor arrangement. (a) shows the abstract layout template with each logical device colored identically among unit devices and (b) depicts the generated layout with the capacitor instances of logical device C4, C10, C16, and C22 being marked.

In order to support design decisions, we included a method into the template that quickly estimates the effective capacitance ratio of the logical capacitors with routing. Based on the areas of both unit devices and routing, estimates of the effective capacitance ratio are calculated (Fig. 5). The influence of routing is estimated by the template using the cumulative routing area per logical device over substrate. In the PDK used, both capacitor area and routing area contribute by about the same amount. Thus, the curves of both device area and routing area over logical capacitor (that should be congruent) can be added. The overall ratio is already acceptable, especially when considering the fast generation speed. Some extensions of the routing options might still be included in order to further improve the result.

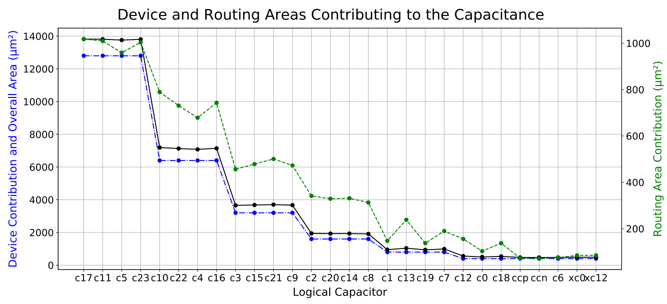

Figure 5 Areas of both capacitors (dash-dotted) and routing (dashed) that contribute to the capacitance ratio (solid). While the capacitors realize the exact target ratio, the routing causes deviations. Still, a good overall ratio is achieved quickly.

V. Conclusion and Outlook

Our presented approach enables accelerated and more flexible generator-based layout creation. It combines a template-based *Layout API* with generators which, in addition, are created automatically through a *Generator Creator*. This novel combination allows flexible array-style layout automation that is user-driven as designers have full control over the placement. In addition, our approach eases generator-based reuse across design projects and process technologies. We validated our methodology using a capacitor arrangement for later use in a SAR ADC. With this, a flexible, template-driven generator is automatically created which both helps to accelerate layout design and eases parasitic estimation (that would not be possible this fast manually). We believe that the combination of templates and generators is a valuable automation approach to meet both flexibility and fast generation speed in a user-driven way.

As next step, we will extend the *Layout API* to support more template types. In addition, we will include more routing options in order to provide higher flexibility and to further extend parasitic trade-off estimations.

Acknowledgment

This work was enabled by the project AnastASICA (grant no. 16ES0990 and 16ES0989) which is funded by BMBF. Many thanks to Christian Albrecht who significantly improved and extended the Generator Creator and many thanks to the reviewers for providing detailed feedback.

References

[1] A. Unutulmaz, G. Dündar and F. V. Fernández, "LDS - A Description Script for Layout Templates," *2011 20th European Conf. on Circuit Theory and Design (ECCTD),* pp. 857-860, 2011.

[2] R. Castro-López, O. Guerra, E. Roca and F. V. Fernández, "An Integrated Layout-Synthesis Approach for Analog ICs," *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems,* vol. 27, no. 7, pp. 1179-1189, 2008.

[3] R. Martins et al., "AIDA: Robust Layout-Aware Synthesis of Analog ICs Including Sizing and Layout Generation," *2015 Int. Conf. on Synthesis, Modeling, Analysis and Simulation Methods and Applications to Circuit Design (SMACD),* pp. 1-4, 2015.

[4] N. Jangkrajarng et al., "IPRAIL—Intellectual Property Reuse-based Analog IC Layout Automation," *Integration, the VLSI Journal,* vol. 36, pp. 237-262, 2003.

[5] P. Pan et al., "A Fast Prototyping Framework for Analog Layout Migration With Planar Preservation," *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems,* vol. 34, no. 9, pp. 1373-1386, 2015.

[6] A. Graupner, R. Jancke and R. Wittmann, "Generator Based Approach for Analog Circuit and Layout Design and Optimization," *2011 Design, Automation & Test in Europe (DATE),* pp. 1-6, 2011.

[7] D. Payne, "A Review of an Analog Layout Tool Called HiPer DevGen," Nov. 28 2011. [Online]. Available: https://semiwiki.com/x-subscriber/tanner-eda/885-a-review-of-an-analog-layout-tool-called-hiper-devgen/. [Accessed 14 05 2021].

[8] E. Chang et al., "BAG2: A Process-Portable Framework for Generator-Based AMS Circuit Design," *2018 IEEE Custom Integrated Circuits Conf. (CICC),* pp. 1-8, 2018.

[9] 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.

[10] T. Reich, U. Eichler, K.-H. Rooch and R. Buhl, "Design of a 12-bit Cyclic RSD ADC Sensor Interface IC Using the Intelligent Analog IP Library," *ANALOG 2013 - Entwicklung von Analogschaltungen mit CAE-Methoden,* 2013.

[11] 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.

[12] A. Krinke, M. Mittag, G. Jerke and J. Lienig, "Extended Constraint Management for Analog and Mixed-Signal IC Design," *2013 European Conf. on Circuit Theory and Design (ECCTD),* pp. 1-4, 2013.

[13] A. Krinke, "Constraint Propagation for Analog and Mixed-Signal Integrated Circuit Design," *Fortschritt-Berichte VDI,* vol. 20, no. 474, Dissertation. 2020.

[14] A. Nassaj, J. Lienig and G. Jerke, "A New Methodology for Constraint-Driven Layout Design of Analog Circuits," *Proc. of the 16th IEEE Int. Conf. on Electronics, Circuits and Systems (ICECS 2009),* pp. 996-999, 2009.

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

[16] B. Prautsch, U. Eichler, T. Reich, A. Puppala and J. Lienig, "Abstract Technology Handling for Generator-Based Analog Circuit Design," *GMM-Fachbericht 83, Reliability by Design (ZuE 2015), VDE Verlag,* pp. 56-61, 2015.

[17] J. Crossley et al., "BAG: A Designer-Oriented Integrated Framework for the Development of AMS Circuit Generators," *Computer-Aided Design (ICCAD), 2013 IEEE/ACM Int. Conf.,* pp. 74-81, 2013.

[18] R. Martins, A. Canelas, N. Lourenço and N. Horta, "On-the-fly Exploration of Placement Templates for Analog IC Layout-Aware Sizing Methodologies," *2016 13th Int. Conf. on Synthesis, Modeling, Analysis and Simulation Methods and Applications to Circuit Design (SMACD),* pp. 1-4, 2016.

[19] B. Prautsch, U. Hatnik, U. Eichler and J. Lienig, "Template-Driven Analog Layout Generators for Improved Technology Independence," *Proc. of ANALOG 2018,* pp. 156-161, 2018.

[20] 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.

[21] A. C. Kammara and A. König, "Absynth: A Comprehensive Approach for Full Front to Back Analog Design Automation," *2018 15th Int. Conf. on Synthesis, Modeling, Analysis and Simulation Methods and Applications to Circuit Design (SMACD),* pp. 165-168, 2018.

[22] A. Unutulmaz, G. Dündar and F. V. Fernández, "A Template Router," *2011 20th European Conf. on Circuit Theory and Design (ECCTD),* pp. 334-337, 2011.

[23] L. Sun, Q. Dai, C. Lee and G. Qiao, "The Analysis on the Parasitic Capacitors Effect of the Fully Differential Architecture of SAR ADC," *Applied Mechanics and Materials, vol. 20–23,* pp. 342-345, 2010.

[24] Y. Zhu, U.-F. Chio, H.-G. Wei, S.-W. Sin, S.-P. U and R. P. Martins, "Linearity Analysis on a Series-Split Capacitor Array for High-Speed SAR ADCs," *VLSI Design,* 2010.

[25] M. J. M. Pelgrom, A. C. J. Duinmaijer and A. P. G. Welbers, "Matching Properties of MOS Transistors," *IEEE Journal of Solid-State Circuits,* vol. 24, no. 5, pp. 1433-1439, 1989.

[26] A. Hastings, The Art of Analog Layout, 2. ed., Pearson Prentice Hall, 2006.

[27] J. Chen, P. Luo and C. Wey, "Placement Optimization for Yield Improvement of Switched-Capacitor Analog Integrated Circuits," *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems,* vol. 29, no. 2, pp. 313-318, 2010.

[28] M. P. Lin, Y. He, V. W. Hsiao, R. Chang and S. Lee, "Common-Centroid Capacitor Layout Generation Considering Device Matching and Parasitic Minimization," *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems,* pp. 991-1002, 2013.

[29] H. Omran, H. Alahmadi and K. N. Salama, "Matching Properties of Femtofarad and Sub-Femtofarad MOM Capacitors," *IEEE Trans. on Circuits and Systems I: Regular Papers,* vol. 63, no. 6, pp. 763-772, 2016.