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

John Wiley Crossley

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

By

John Wiley Crossley

A dissertation submitted in partial satisfaction of the

requirements for the degree of

Doctor of Philosophy

in

Engineering – Electrical Engineering and Computer Sciences

in the

Graduate Division

of the

University of California, Berkeley

Fall 2013

Abstract

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

by

John Wiley Crossley

Doctor of Philosophy in Engineering – Electrical Engineeing and Computer Sciences

University of California, Berkeley

Professor Elad Alon, Chair

The recent trend in embedding multiple applications into a single System-on-Chip (SoC) has resulted in an increase in the number of Analog/Mixed-Signal (AMS) components integrated per die. Although the AMS components typically occupy a small fraction of the whole IC, they often require the longest design time because typical AMS design flows require substantial manual intervention from the designer throughout 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. In this thesis, we present the Berkeley Analog Generator (BAG) framework, an integrated framework for the development of generators of AMS circuits.

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 – at the schematic and particularly at the layout level – to aid the designer in developing truly parameterized and technology-independent AMS circuit generators. The BAG framework simplifies and helps codify common tasks in the AMS design flow including technology characterization, schematic and testbench translation, simulator interfacing, physical verification and extraction, and layout creation. BAG addresses one of the most labor-intensive tasks, layout, by providing template-based extensible layout generators for different styles of circuits to help designers create their own parameterized layout generators. In order to demonstrate the completeness of the BAG framework, the development process of several generators for an integrated switched-capacitor (SC) regulator and its associated subcircuits are presented as a case study and the top level SC regulator generator is used to create three instances of a SC voltage regulator targeting different power densities, absolute output power, and aspect ratios.

Table of Contents

List of Figures

List of Tables

Acknowledgements

Many people contributed to the work presented in this thesis both directly (through work on the BAG project) and indirectly (through their influence on and connections to me) and I would like to take some space to express my gratitude to them.

First, I would like to thank my advisor, Prof. Elad Alon. A fellow student and I were walking back from dinner with Elad during a tapeout period and the topic of discussion turned towards what we would do in life if we could choose anything and earning money and aptitude for the activity were non-issues. I couldn't come up with anything specific but after having been in tapeout mode for several weeks I said it would definitely be something more fun and less exhausting than designing circuits. The other student replied that he would probably like to play basketball all day. I agreed that would be a good choice and then we both looked towards Elad. He didn't have to say anything but I think all three of us knew the answer. There is nothing Elad would like more than researching, teaching, speaking, and writing about a topic he loves. There is no better quality that a student could hope to find in an advisor and there is no better advisor a student could hope to find than Elad.

I would also like to thank all of the faculty at Berkeley that have helped me during my time here. I am particularly grateful to Prof. Ali Niknejad for serving on my quals committee, reading this thesis, and for teaching the two most challenging circuits classes I ever took. I am indebted to Prof. Bernhard Boser for the advice that led me to join Elad’s group and to Prof. Paul Wright for agreeing to read this thesis at the last minute. I would also like to thank Prof. Vivek Subramanian and Dr. Reza Moazzami for teaching classes that were so interesting that I signed up to take one of their other courses (which wasn’t initially my plan).

The Berkeley Wireless Research Center (BWRC) has been a big part of my life at Berkeley and I would like to thank all of the faculty and staff there. I especially appreciate those who helped me wrestle with CAD tools and computers: Brian Richards, Ken Tang, Kevin Zimmerman, Brad Krebs, and Ubirata Coelho.

I would also like to thank all the advisors and administrators who made my experience at Berkeley as smooth and stress-free as possible: Ruth Gjerde, Shirley Salanio, Olivia Nolan, Leslie Nishiyama, and Tom Boot.

I worked with many fellow students over the years for research and/or class projects and I would like to thank some of them (in no particular order): Eun Ji An, Bonjern Yang, Nicholas Sutardja, Kwangmo Jung, Lingkai Kong, Simone Gambini, Rikky Muller, Lingkai Kong, Yue Lu, Seobin Jung, Chintan Thakkar, Jaeduk Han, Hanh-Phuc Le, Rachel Nancollas, Alberto Puggelli.

Finally, I would like to thank Megan for sticking with me through everything. It is entirely her doing that I have now accomplished what I’m sure will be one the proudest achievements of my life: making it through a Berkeley PhD with my sanity and relationship in tact.

Chapter 1 – Introduction

1.1 The case for analog design automation

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 the core functionality in such integrated systems is often implemented with digital circuitry, some functions that are required to guarantee system functionality and performance (e.g. radio transceivers, temperature sensors, voltage regulators, high speed IO, PLLs) rely on AMS design techniques. Compared to digital circuit design flows, AMS circuit design lacks a common set of well-defined steps in the design flow or the ability to capture a design procedure in an executable way. As a result, even though digital circuits dominate in terms of their total die area – as shown in the A7 SoC in Figure 1.1 – AMS circuits are increasingly the gating factor preventing designs from being completed on time and/or within budget.

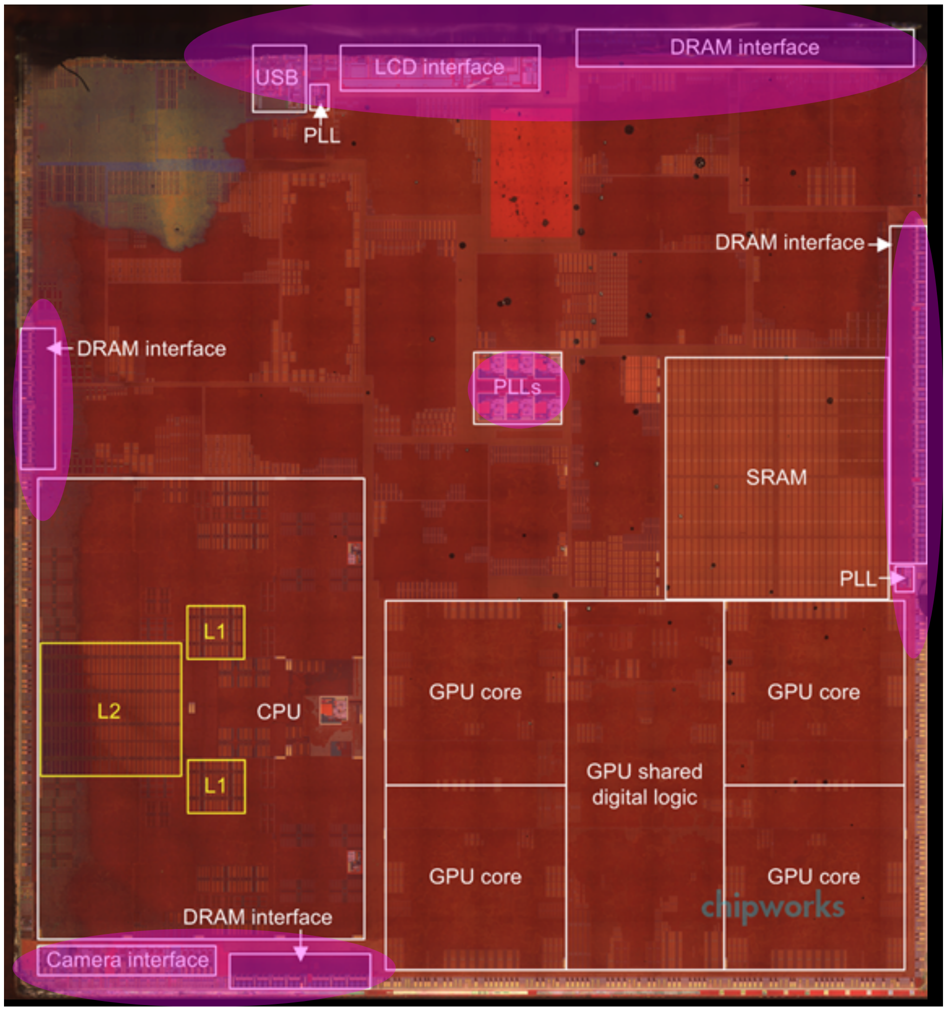


Figure 1.1: Apple A7 SoC with some of the AMS blocks highlighted [2].

As shown in the example AMS design flow of Figure 1.2, AMS designers are directly involved in many steps of the design process, and are also critical in connecting the various steps into a cohesive design flow. While some individual steps – like SPICE simulation and parasitic extraction – do not require a *human-in-the-loop*[1], others (e.g. custom layout) require extensive manual labor. Even the automated steps still require designers to interpret the results, modify design parameters, and pass information to other steps of the design. In contrast, digital design flows centralize all manual efforts into a set of design scripts. The steps of the digital design process, all of which are highly automated, are then connected and controlled from the script rather than directly by the designer. Once a design script has been fully debugged, it is usually possible to modify the inputs (e.g. to modify register transfer language (RTL) descriptions or update the standard cell libraries) and successfully rerun the design procedure without requiring additional work from the designer.

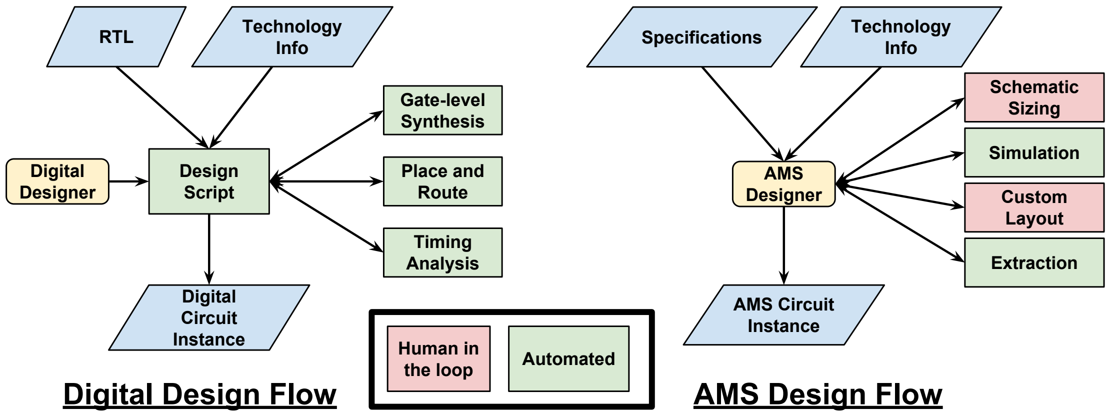


Figure 1.2: Digital vs. AMS design flow.

Today's AMS design flows unfortunately do not have the same property. In fact, the amount of human intervention required for AMS designs is actually increasing due to the increasing interaction between the layout design steps and circuit performance that occurs as designs are started in or migrated to more advanced technology nodes. Creating and verifying "high-quality" (i.e., meeting performance/power as well as reliability/yield constraints) layouts is perhaps the largest single task in a typical AMS circuit design. Unlike digital flows, the layout designer usually still hand-places devices and draws polygons for routing. If manual layout were just a matter of a single continuous process, one time for every tapeout, without requiring any modifications or restarts along the way, it might not be such a big time sink. Unfortunately that situation almost never occurs. Unless the designer has very accurate initial layout parasitics, it is usually the case that after the first-pass layout is completed and the circuit is simulated to include the extracted parasitics, the whole design sizing must be performed again. This of course, leads

[1] SPICE circuit simulation takes a netlist and a testbench script as inputs and produces as output calculated voltages and currents for the nodes and branches of the circuit. While human involvement may be used to produce the inputs and examine the outputs, the process of solving the differential equations is fully automated. Designers tend to take this kind of automation for granted because the amount of effort for the equivalent manual procedure is so high that it is not considered a viable option. Layout parasitic extraction is similarly automated and accepted as the standard.

to changes in the layout, and if those changes are significant enough, it can require almost as much time as the first pass layout. In some cases it might even be more if the designer realizes that they neglected certain constraints during the first pass. All of this effort is only assuming a normal layout procedure and not accounting for any of the numerous layout-affecting scenarios that could result from a change in the specifications or a change to the process.

Given the labor-intensive nature of manual AMS circuit design, it would thus be highly desirable to automate the design of AMS circuits, in a manner analogous to automated digital design techniques, and foster their reuse across multiple SoCs and technology generations. This would shorten the time-to-market of new products, and would free analog designers from performing repetitive tasks (e.g. circuit redesign due to a change in the specifications or technology migration).

In order to understand the advantages that design automation can bring to AMS circuit design, it is useful to examine a representative example design that uses the traditional, manual design flow. The digital phase locked loop (DPLL) shown in Figure 1.3 is one implementation of a clock generator – which is a common AMS circuit block – that was created using a manual AMS design flow. The goal of this design was to generate a low-jitter clock in the presence of significant supply noise with minimal power consumption [3]. The most unique parts of the design were the overall architecture combining three different control paths, and the circuit techniques used to control the phase of the DCO. Most of the sub-blocks involved – such as the current DAC, capacitor DAC, low-pass filter, and phase frequency detector – used standard architectures and well-understood design procedures. Despite this, a large portion of the total work that went into the design was spent on these blocks, which although they were necessary for the correct function of the circuit, were not critical to the main advances we intended to demonstrate with the PLL architecture. From an academic perspective, automating these types of circuits could allow research to be more focused and allow less time to be spent implementing excessive amounts of ancillary circuits just to demonstrate one new idea. Automating designs like these is also attractive from an industrial perspective in order to simplify and speed up the design process as well as to improve the productivity of AMS circuit designers.

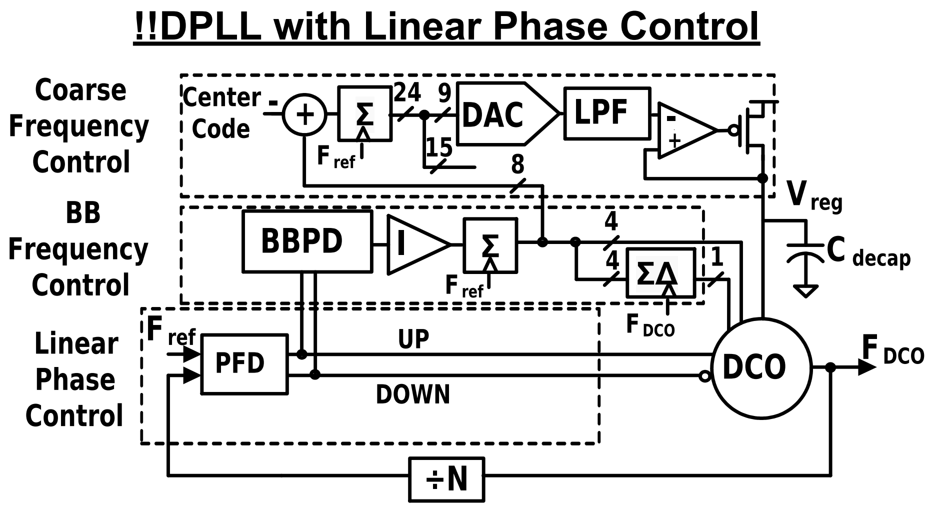


Figure 1.3: Digital PLL Block Diagram.

Ancillary circuits are not the only type of circuits that could benefit from automation. New architectures and design techniques could spread faster and have wider impact if designers had more resources other than research papers and design review slide decks. In Figure 1.3, the linear regulator used to filter out supply noise in the digital PLL made use of design techniques from recent work on replica compensated regulators [4]. In contrast to the previously mentioned circuits, the regulator had a more direct impact on the jitter performance and supply rejection of the PLL and the chosen regulator architecture was less widely known (and more importantly, not known to the author at the time, who was the PLL designer). A research paper can be helpful when learning how to design a new circuit, but if that is the only resource available, it leaves a lot of room for misunderstanding on the part of the reader and miscommunication on the part of the writer. Having an automated design procedure for the regulator that could be examined and rerun with different parameters, even if the procedure required modifications in order to reuse it in the context of the PLL, would have been very useful in understanding the circuit, speeding up the design process, and achieving the best performance.

So far, the digital PLL example has been used to motivate the general notion of AMS circuit design automation without suggesting any specific features that should be included in an automated design flow. One feature that would have been particularly useful in the final days before the tape-out deadline of the digital PLL is a degree of programmability in the aspect ratio of the layout. Figure 1.4 shows two top-level layouts for the digital PLL that were separated in their time-of-completion by two days. The layout on the left, which measures 1x1 mm and is pin-limited, had just been completed when the foundry sent notification that the shuttle run only had room for chips with one dimension smaller than 450 um. The layout on the right was completed two days later (just in time to meet the deadline) after considerable manual layout modifications. If the foundry had specified 400 instead of 450 um, the manual layout modification effort would have been too large to meet the deadline because it would have required modification of the PLL core itself and not just the bond pads, power grid and top level routing. An automated layout procedure that could target a particular aspect ratio or constrain a single dimension of the layout would have worked in both scenarios and would have had the added benefit of giving the designer (who once again, was the author of this thesis) a lot more sleep and a lot less stress over the final two days of the tapeout period.

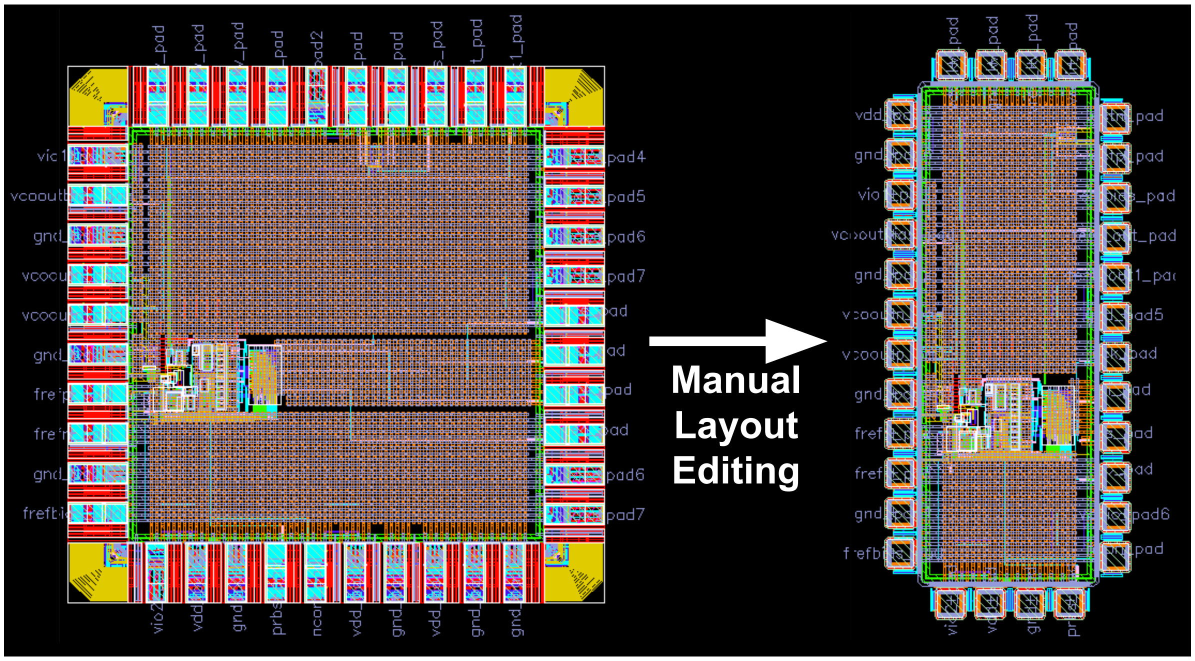


Figure 1.4: Digital PLL chip layout before and after a last minute change to the allowable dimensions.

1.2 Previous Work

The idea of automating analog circuit design is by no means novel, and significant effort using a variety of methodologies has been made to automate various steps of the AMS circuit design flow. In [5], Gielen and Rutenbar give an excellent overview of many of these efforts, which they divide into two broad categories: "knowledge-based techniques" and "optimization-based techniques". In knowledge-based automation techniques [6], [7], design steps tailored to specific circuit architectures (e.g. Flash ADC, Switched-Capacitor filters, etc.) are encoded into a design procedure, i.e. scripts that replicate the activity of the designer. 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 flow, thus easing modifications and debugging. On the other hand, the activity of setting up the scripts can 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 [8], [9], in contrast to knowledge-based techniques, 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 properly constrain the optimizer. 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 these efforts, the AMS circuit 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 on 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 initial effort required to 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 their central role (and perhaps their 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 improved the tools that can help automate individual steps in the analog CAD design flow.[2] In the following chapters, we describe our effort towards enabling a widely-adopted shift in the methodology used to design AMS circuits through the creation of a design framework capable of codifying all the steps of the design flow, the development of helper classes to ease (from the designer's perspective) the process of codifying the schematic sizing and especially the layout procedure, and the creation – using the aforementioned framework and helper classes – of circuit generators for a non-trivial circuit capable of creating multiple instantiations targeting different input specifications.

1.3 Organization

Chapter 2 introduces the Berkeley Analog Generator (BAG) framework, 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. A definition of an AMS circuit generator is provided followed by an explanation of the goals of the BAG project, a description of a generator design procedure, and details of the framework's implementation. Chapter 3 focuses on the BAG framework's approach towards automating layout by enabling designers to create layout generators for specific circuit architectures. In order to assist the designer in creating layout generators, the BAG framework provides template-based extensible layout generators for two styles of parameterized layout, suited respectively to heterogeneous and homogeneous AMS circuits. Chapter 4 presents a case study for a switched-capacitor converter generator that is used to create three converter instances targeting different output power levels and efficiencies as well as different physical aspect ratios. Finally, we conclude and discuss plans for the future of the BAG framework in Chapter 5.

[2] In fact, CAD research in the field is still active, both in the academic and industrial worlds [10]–[13]

Chapter 2 – Berkeley Analog Generator Framework

In this chapter, 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 goals of closing the gap between the designer and CAD communities, and providing designers with a framework that allows them to take advantage of the CAD tools available to help automate their designs and foster reuse. Designers can use BAG to develop circuit architectures closely following all steps of their familiar custom flows. 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 that 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 with varying 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.

2.1 What is an AMS circuit *generator*?

An AMS circuit *generator* is a set of design functions that can create a fully sized schematic and a layout of a circuit according to a set of input specifications. Any such generator should, at a minimum, be capable of reading a set of input specifications, automatically sizing all schematic components, generating a corresponding physical layout that meets all design rules and Layout Versus Schematic (LVS) checks, and outputting the resulting circuit performance, 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) [14], we thus define the *interface* Analog Generator shown in Figure 2.1. Each circuit generator can be implemented as a *class* that has to implement all the methods specified in the Analog Generator interface. Although individual generators 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. An AMS circuit generator framework can thus provide a set of *abstract base classes* to help a designer create new generators by providing interfaces to tools that perform common functions in the design process. Figure 2.1 shows an example of some of these functions as part of a sample abstract base class from which all analog generator classes could inherit.

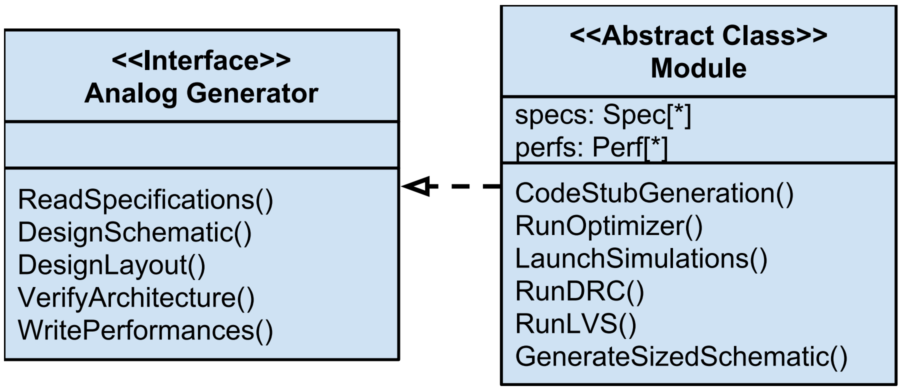


Figure 2.1 : AMS circuit generator interface and an example abstract class implementation of the interface.

In our approach to analog circuit design 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. These generators can replicate, in an automated fashion, the design procedure that would have been used for a traditional, manual design.

2.2 Goals for the BAG Framework

The primary goal of the BAG project is to provide a framework that encourages and enables circuit designers to become circuit generator designers. To that end, given the poor adoption rate of existing automation tools, there are several important aspects that we try to address in order to create a framework that designers can and will actually use. These aspects, which will be described further in this section, are: completeness, ease of adoption, and ease of reuse.

2.2.1 Completeness

An automation framework that only focuses on a portion of the design procedure can be very useful, but is much less likely to be adopted in the long run. Many of the key benefits of automation, including the ability to rapidly adapt and reuse pre-existing designs and perform quick design iterations, are precluded if the framework requires substantial manual effort for even a single design step. As an example, the performance of AMS circuit designs, especially those made in today's advanced technology nodes, are greatly impacted by the effect of layout parasitics. Therefore, including some form of layout automation in the framework is vital in order to enable meaningful design iterations that do not require massive amounts of manual tweaking, or worse – complete redesign – when translated from schematic to layout.

Remaining comprehensive in the face of advancements in design tools depends on the framework being extensible. One way to ensure a framework remains complete is to hire a group of developers to maintain and enhance the framework. Another way, which is much more feasible for an academic project, is to allow the same community that uses the framework to modify and improve the source code of the framework itself. In order for this method to work, the framework must be adopted by a large enough number of people so that the small fraction who actually contribute useful code are sufficient to cover a large variety of design techniques and tools necessary for a truly comprehensive framework. One way to encourage this is to release a framework's source code[3] under an open source license using a collaborative source code revision control tool such as GIT [16]. This doesn't guarantee that people will use the framework or contribute improvements or bug fixes, but at least it enables the widest possible audience and hopefully improves the odds of finding enough contributors to maintain the framework.

2.2.2 Ease of Adoption

Conventional wisdom says you never get a second chance to make a good first impression. The first impression a new user forms of a design system highly depends on the effort, and especially the amount of time, required to start using the system effectively. The more difficult and the longer this time period, the more the designer will resist trying the tool and the less likely they will be to adopt it in the long term. In order to minimize this time, an AMS automation framework must build on the base-level of knowledge and common skill-sets possessed by most AMS designers, and must do so in a way that seems somehow familiar to the designer.

Allowing designers to codify their existing design procedures, rather than forcing them to conform to a specific design style will let designers more readily adapt to a generator-centric design methodology. If they have to learn a new or unfamiliar design style, whether it be optimization-based or knowledge-based, in addition to a new framework of tools for executing the design process, they will then require more time to adapt which might provide an excuse to dismiss the tool completely.

Another way the framework can reduce the initial learning curve is to provide an intermediate step between a manual, hand-tweaked circuit design and a fully scripted circuit generator. An interactive, console-based script interface can serve as this intermediate step by allowing the designer to explore the design manually using traditional techniques but still using the same functions that will eventually be used for the fully scripted generator. In this way, the task of writing the generator code can be made somewhat less difficult as the designer can adapt code fragments from the command history of the interactive session and there can be less duplication of design effort.

[3] A pointer to the BAG framework source code can be found [15]

2.2.3 Ease of Reuse

While it is important to reduce the difficulty of the designer's first experience with writing a generator, it is equally important that their subsequent experiences offer substantial time savings compared to standard design practices, otherwise the designer will have no incentive to continue using it. To that end, an AMS automation framework must encourage and enable simple reuse mechanisms. A common scheme used to simplify the reuse of a complex codebase is object-oriented programming (OOP) [17].

There are many resources available that expound at length on the topic of object-oriented programming [18], [19], so this section will only present the handful of OOP concepts that are particularly useful to the task of creating a reusable AMS automation framework. One of these concepts is *encapsulation*, which dictates that data and the procedures that operate on that data are grouped together in self-contained bundles called *objects*. *Objects* of the same type, i.e. containing the same categories of data (though not necessarily the same data) and the same operations, are said to be *instances* of a particular *class* of objects. An *object* is a natural representation for a circuit model since a circuit is something that has certain attributes, e.g. connectivity and sizing, but also performs a particular operation amongst inputs and outputs. In order to create a design framework for circuit generators, we merely have to extend the concept of a circuit as an *object* to include all the operations and data necessary for the design procedures that produce an instance of a circuit instead of just the end result. OOP also enables direct reuse of code by enabling *classes* to *inherit* functionality and data in a hierarchical fashion from higher-level classes. This concept can easily be applied to circuits that belong to the same class to provide certain functions that are shared by all circuits of that class such as simulation testbenches, e.g. a phase noise simulation testbench that works for both LC and ring oscillators.

A final factor impacting the reusability of generators designed using a particular framework is the structure of the framework itself. If interfaces are defined inappropriately, it can be difficult to encapsulate the shared functionality of the generators. Similarly, if circuit generators are created at too high a level within the circuit hierarchy, there won't be very much reuse. As an extreme example, if a PLL was created as a single monolithic generator made up of primitives, i.e. no sub modules are themselves generators, then nobody would be able to reuse the charge pump or VCO from the PLL for other circuits without substantial modification to the generator code.

2.3 Framework Implementation

A set of abstract goals is not sufficient information to begin working on a complex codebase. Before an actual generator framework can be created, several questions must be answered: What will the design flow for creating a generator for a new circuit architecture look like? What programming languages and tools will be needed to build a framework that enables this design flow? How will the framework be organized? This section will answer these questions with regard to the development of the BAG framework and also describe a set of useful *helper classes* that, while not required to implement the *interface* defined in Figure 2.1, help the BAG framework achieve some of the goals of the previous section.

2.3.1 Generator Design Flow Description

Although the end product is a collection of code, as shown in Figure 2.2, the BAG design flow 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 [20]. 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* or *low power*.

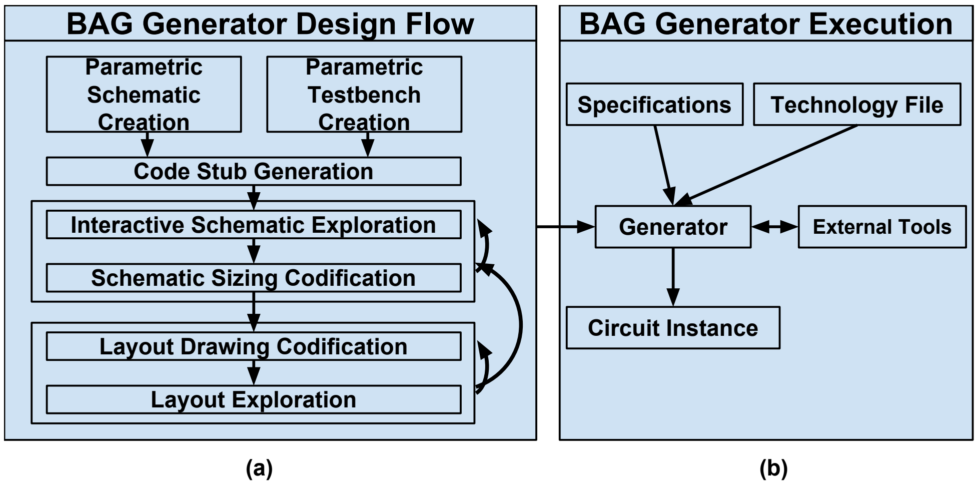

BAG Generator Design Flow
Parametric Schematic Creation
Parametric Testbench Creation
Code Stub Generation
Interactive Schematic Exploration
Schematic Sizing Codification
Layout Drawing Codification
Layout Exploration
BAG Generator Execution
Specifications
Technology File
Generator
External Tools
Circuit Instance
(a)
(b)

Figure 2.2: Design flow used to produce a generator using BAG (a) and the generator execution process to produce a circuit instance (b).

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, and probe points -- through Cadence ADE [13]. Testbench schematics in BAG are atomic, i.e. only one simulation per testbench, which improves their reusability. The parameterized design and testbench schematics are imported recursively in BAG and used to create stub class definitions, which inherit from the *DesignModule* and *TestbenchModule* classes respectively, for every unique cell in the hierarchy.

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 next steps of the design flow. 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. Ultimately, however, the process independence of a generator does depend on the designer.

Once the initial classes for the design under development and its associated testbenches are created, and an initial process technology has been selected, the designer can then – 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 to lowering the designer effort in writing the generator code. For example, snippets of code used in the exploration process can be used to build *DesignSchematic()* and *VerifyArchitecture()* methods that are required to implement an Analog Generator interface, as defined in Figure 2.2.

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 a *DesignLayout()* function. 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 then be added to a *VerifyArchitecture()* function. 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. Given the importance of these effects on designs in modern technologies, it can be useful to perform the layout generator creation step earlier – before spending much time on the *DesignSchematic()* function – since any design procedure developed without reasonable estimations of the layout effects would need to be modified later in any case.

After the initial pass through the design flow, the designer can iteratively refine the generator class to improve the design's performance, e.g. by calling numerical or equation based optimizers, and/or 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 Figure 2.2.

2.3.2 Language and Tools

Several platforms were considered for implementing the BAG framework, including Matlab, Perl, Python, SKILL and Tcl. Table 2.1 shows a mostly subjective comparison of these languages. For a more objective analysis, see [21]. All of the languages have some strong points. SKILL is a proprietary language tied strongly to Cadence's Virtuoso, the most popular existing analog CAD tool suite, but it lacks the breadth of functionality available with most of the other languages. Perl has a lot of useful libraries available and is quite fast, but is notoriously difficult to read. Tcl is a popular choice in electronic design automation (EDA) tools and is generally easier to understand than Perl, but it lacks speed and has fewer useful libraries. Matlab is easy to use and has an excellent interactive shell, but it is also somewhat slow, and its object-oriented features are less mature than many of the other languages.

Ultimately, Python was chosen as the implementation language for several reasons. It has strong support for object-oriented features such as abstract base classes and multiple inheritance which can help achieve the desired goals of easy extensibility and reuse. Additionally, Python's indentation-based structure and highly readable syntax are well suited for inexperienced or novice programmers[4]. It is also fast, both in terms of execution time and the amount of time required to create working code.

	TCL	**Python**	**Perl**	**Matlab**	**SKILL**
Code Readability	Good	Excellent	Bad	Good	Fair
OOP	Tacked On	Built in	Tacked On	Tacked On	Tacked On
External Libraries	Fair	Excellent	Good	Good	Bad
Interactive Shell	Yes	Yes	No	Yes	Yes
Speed	Fair	Good	Good	Fair	Fair

Table 2.1: Subjective comparison of scripting language choices for creating an AMS circuit generator framework.

More expressive generators can be written by leveraging the plethora of off-the-shelf packages provided in Python, ranging from numerical and scientific computing (NumPy [22] and SciPy [23]), to graphical plotting (Matplotlib [24]), and numerical optimization (e.g., CVXPY [25]). Python also has a powerful interactive command line that can enable generator designers to rapidly iterate during the generator codification process. The default python console provides most of this functionality, but there are other add-ons for python, namely Ipython [26], that offer even more integration of the aforementioned tools. Ipython provides a *pylab* mode that integrates NumPy, SciPy, and Matplotlib functionality into the console along with improved code completion and debugging tools. Figure 2.3(a) shows a plot result from a simulation run from the pylab console. Recent versions of Ipython introduce a *notebook* mode, similar to Matlab's *cell* mode [27], where scripts can also be written in an active document format that can be edited and run from a web browser. Results, in the form of standard output text and inline graphics can be saved along with the script itself. This format, show in Figure 2.3(b), can be useful for debugging scripts because the user can rerun certain cells without rerunning the whole script. It is also useful for recording the results of a scripting session along with detailed documentation that can be inserted inline with the code using a simple, wiki-like markup language.

[4] Increasingly, AMS circuit-designers coming out of school have a substantial amount of practical programming experience, so this point is less critical than it may have been in the past.

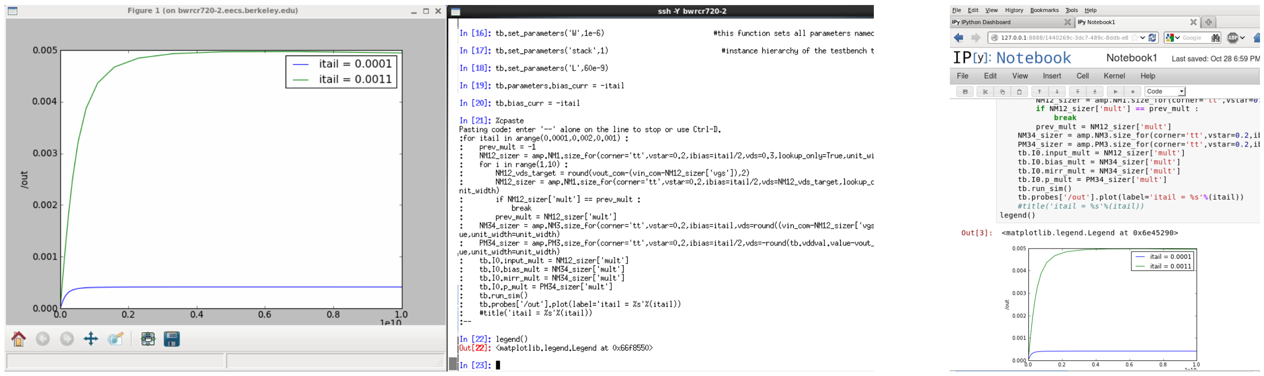


Figure 1 (on bwrcr720-2.eecs.berkeley.edu)
itail = 0.0001
itail = 0.0011
/out
1e10
ssh -Y bwrcr720-2
File Edit View History Bookmarks Tools Help
IPython Dashboard
Notebook1
127.0.0.1:8888/1440269c-3dc7-489c-8ddb-e8
IP[y]: Notebook
Last saved: Oct 28 6:59 PM
File Edit View Insert Cell Kernel Help
Code

```
In [16]: tb.set_parameters('W',1e-6)                       #this function sets all parameters namec

In [17]: tb.set_parameters('stack',1)                       #instance hierarchy of the testbench t

In [18]: tb.set_parameters('L',60e-9)

In [19]: tb.parameters.bias_curr = -itail

In [20]: tb.bias_curr = -itail

In [21]: %cpaste
Pasting code; enter '--' alone on the line to stop or use Ctrl-D.
:for itail in arange(0.0001,0.002,0.001) :
:    prev_mult = -1
:    NM12_sizer = amp.NM1.size_for(corner='tt',vstar=0.2,ibias=itail/2,vds=0.3,lookup_only=True,unit_wi
:    for i in range(1,10) :
:        NM12_vds_target = round(vout_com-(vin_com-NM12_sizer['vgs']),2)
:        NM12_sizer = amp.NM1.size_for(corner='tt',vstar=0.2,ibias=itail/2,vds=NM12_vds_target,lookup_c
nit_width)
:        if NM12_sizer['mult'] == prev_mult :
:            break
:        prev_mult = NM12_sizer['mult']
:    NM34_sizer = amp.NM3.size_for(corner='tt',vstar=0.2,ibias=itail,vds=round((vin_com-NM12_sizer['vgs
ue,unit_width=unit_width)
:    PM34_sizer = amp.PM3.size_for(corner='tt',vstar=0.2,ibias=itail/2,vds=-round(tb.vddval.value-vout_
ue,unit_width=unit_width)
:    tb.I0.input_mult = NM12_sizer['mult']
:    tb.I0.bias_mult = NM34_sizer['mult']
:    tb.I0.mirr_mult = NM34_sizer['mult']
:    tb.I0.p_mult = PM34_sizer['mult']
:    tb.run_sim()
:    tb.probes['/out'].plot(label='itail = %s'%(itail))
:    #title('itail = %s'%(itail))
:--

In [22]: legend()
Out[22]: <matplotlib.legend.Legend at 0x66f8550>

In [23]:
```

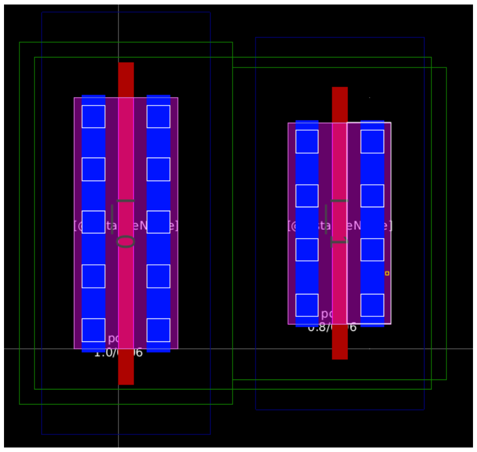
```
        if NM12_sizer['mult'] == prev_mult :
            break
        prev_mult = NM12_sizer['mult']
    NM34_sizer = amp.NM3.size_for(corner='tt',vstar=0.2,ib
    PM34_sizer = amp.PM3.size_for(corner='tt',vstar=0.2,ib
    tb.I0.input_mult = NM12_sizer['mult']
    tb.I0.bias_mult = NM34_sizer['mult']
    tb.I0.mirr_mult = NM34_sizer['mult']
    tb.I0.p_mult = PM34_sizer['mult']
    tb.run_sim()
    tb.probes['/out'].plot(label='itail = %s'%(itail))
    #title('itail = %s'%(itail))
legend()

Out[3]:  <matplotlib.legend.Legend at 0x6e45290>
```

(a) **(b)**

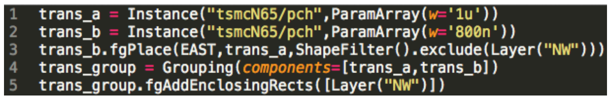
Figure 2.3 : Ipython plot window and console (a) and browser-based notebook interface (b).

The choice of the Python platform is further supported by the availability of Synopsys' PyCell framework, which provides a (freely-available) API for creating process-independent parameterized layout cells (PCells) using ``DRC correct-by-construction" functions. As will be described in more detail in the next chapter, PyCells are used as the basis for automating layout in the BAG framework. At design time, PyCell API functions can be used to concisely express the desired relative placements of geometric objects. The enforcement of DRC correctness is postponed until 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 the PyCell API will automatically enforce a process-specific distance between the two devices such that the DRC rules for all layers in the two devices are met. By thoughtfully building up hierarchical geometries using these functions, it is possible to create PyCells that can be compiled for different technology nodes. Figure 2.4 shows an example of the PyCell API placing two PMOS transistors next to each other as close as possible on all layers excluding the nwell layer, and then enclosing both transistors within an nwell rectangle sufficient to meet the minimum enclosure rules for the nwell layer.

(a)



(b)



```
trans_a = Instance("tsmcN65/pch",ParamArray(w='1u'))
trans_b = Instance("tsmcN65/pch",ParamArray(w='800n'))
trans_b.fgPlace(EAST,trans_a,ShapeFilter().exclude(Layer("NW")))
trans_group = Grouping(components=[trans_a,trans_b])
trans_group.fgAddEnclosingRects([Layer("NW")])
```

Figure 2.4: Synopsys PyCell layout (a) resulting from correct-by-construction operations (b).

The PyCell API is also part of the interoperable process design kit (iPDK) specification [28]. The iPDK is an attempt by a large group of EDA vendors and foundries (most notably TSMC), to create an interoperable custom design framework. It is built around the already widely-adopted OpenAccess [29] design database and uses PyCells to provide parameterized layout cells. At the time of the writing of this dissertation, it seems that the iPDK, and through it the PyCell API, is seeing increasing adoption. Correct-by-construction operations are extremely useful, but if no foundry offers a PDK that supports PyCells, then it wouldn't make sense to include them in the framework.

Although support for open standards is not a primary goal of the BAG framework, we tried to choose open tools and APIs whenever possible to ensure the most interoperability and reduce dependence on proprietary standards that can change at the whim of a single EDA company. The code for the Synopsys tools used for creating PyCells, while not open source, is held in escrow with the condition that, should Synopsys ever stop offering it free of charge or providing support, the source will be released to Si2, the industry organization responsible for maintaining the OpenAccess API. We also tried to give the designer freedom of choice in terms of design tools by identifying the core functionality required for the design flow and defining a standard interface between that core functionality and the specific tools used to provide that functionality. In practice, although we have tried to define a general interface that could enable any EDA tool to be incorporated into the BAG framework, in most cases we have initially only enabled a single tool for each piece of the design flow.

2.3.3 Code Organization

The codebase for the BAG framework can be split roughly into two groups of classes as depicted in Figure 2.5. The *generator description* group contains classes used to express the design itself, including the physical connectivity, device sizes, and the design procedure. In some sense, these classes provide all the functionality needed for a circuit designer to write a generator. What they lack is the ability to *generate* a specific instance of a circuit. The *generator session* group contains classes that provide interfaces to all the external tools and process information that are necessary to execute a *generator* and produce a circuit instance in a given technology. The central classes[5] for the *generator description* and *generator session* groups are *Module* and *BagProject* respectively. This section will describe the division of functionality and design data amongst these, and other, classes.

[5] These classes are important to making a functional framework and so need to be understood, but they do not present particularly interesting challenges. The more compelling contributions of this work are the helper classes covered in 2.3.4 and the layout styles in Chapter 3.

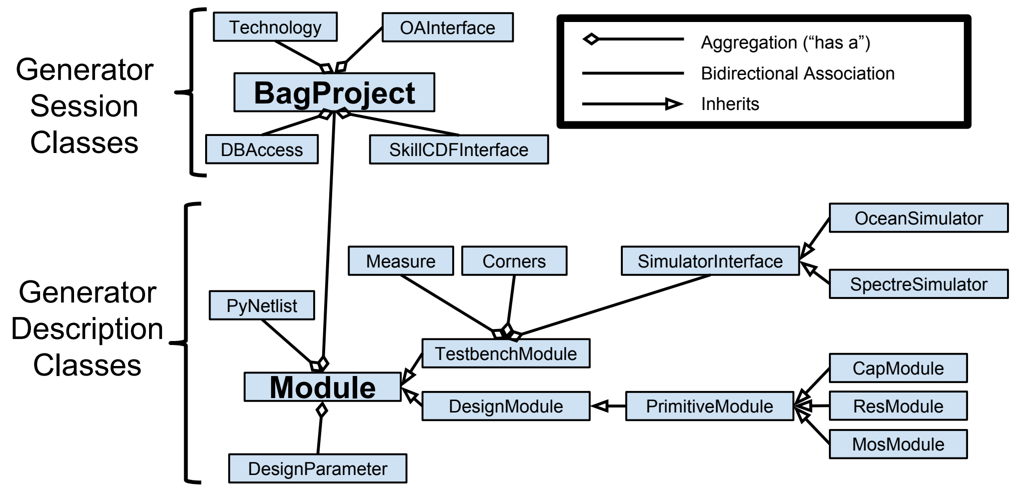


Figure 2.5 : BAG class structure UML diagram.

2.3.3.1 Generator Session Classes

We define the *generator session* as the process of executing a generator to produce an instance of a particular circuit. The purpose of the *BagProject* class is to store information related to the *generator session* as well as interfaces to external tools required during the *generator session* in a central location that can be easily referenced from anywhere in the generator code. It is important that these interfaces not incur excessive execution-time overhead, i.e. on top of the inherent overhead of the tool being interfaced. This section will describe further the primary child classes of *BagProject*: *DBAccess*, *DBInterface*, *CDFInterface*, and *Technology*.

2.3.3.1.1 DBAccess

Many of the external tools interfaced by the BAG framework belong to the Cadence Virtuoso [13] design suite. The purpose of the *DBAccess* class is to provide a single interface point for all the classes in the BAG framework that need to access Cadence tools such as those for setting CDF parameters (*CDFInterface*) and those for running the Cadence simulation environment (*OceanSimulator* and *SpectreSimulator*). There were several alternatives for implementing this interface. One option to transfer data between Cadence and the BAG framework was to wrap the Cadence Integrator's Toolkit C API with Python using SWIG [30] or a similar automated software interface generator. We chose not to do this because the integrator toolkit only provides functions for accessing the design database, and not for running the full set of tools in the design suite such as the Spectre circuit simulator. Another option was to launch specific Cadence executables using python's *os.system()* function and redirect the output to a file for eventual parsing. We also rejected this method for two reasons. First, it blocks the execution of subsequent python code until the executable process exits. Second, it leads to wasted time loading and reloading the executable. The full Virtuoso executable can take several seconds to load so if the BAG framework calls low-level – i.e. usually quick execution time – functions from Cadence, the overhead of launching and killing the executable process becomes unacceptably large. In order to avoid this penalty, the BAG framework launches a single

Cadence process and leaves it open during the whole design session, passing information back and forth between the framework and Cadence as necessary.

The communication between the BAG framework's *DBAccess* class and the Cadence executable is facilitated by the *Pexpect* module [31]. *Pexpect* is used to spawn an interactive console-based process and control it through code using a query/response interface rather than through direct user input. *Pexpect* allows the BAG framework to have a virtual Virtuoso console running in the background in order to run SKILL commands and scripts. It also allows non-blocking execution so the process can continue running in the background for the whole design session. Other classes that require access to Cadence tools contain a link to the *generator session*'s *BagProject* object and consequently a link to the *DBAccess* object and Cadence executable.

2.3.3.1.2 DBInterface & OAInterface

Custom circuit designs are typically stored in a database that groups together the schematic, symbol, layout, and other data regarding a particular cell into a single library entry. Since the BAG framework is intended to be as tool-agnostic as possible, an *abstract base class* called *DatabaseInterface* defines a generic interface to a design database. The two main design databases are the OpenAccess and the CDBA databases. The BAG framework currently only has a subclass for the OpenAccess database called *OAInterface*. OpenAccess was chosen over CDBA as the initial database implementation because it is non-proprietary and used by all the major custom circuit design suites from the top EDA vendors.

The *OAInterface* class has functions used to instantiate layout instances used for DRC, LVS, and extraction. It also has functions to help create sized schematics by copying and modifying template schematics. *OAInterface* depends on a python module [32] that has python wrappers for the OpenAccess C++ API[6]. This functionality could also have been provided by the Cadence Integrator's toolkit C API, but that would have required extra effort to wrap the C API with Python. Another option was to use Cadence's SKILL API for modifying the OpenAccess database, but we chose using the OpenAccess API directly to remain as tool-agnostic as possible.

2.3.3.1.3 CDFInterface & SkillCDFInterface

The OpenAccess database, though quite extensive, neglects to provide a standard mechanism for storing and modifying the user-settable parameters associated with schematic cells or for the callback functions associated with these parameters. These parameters, or common description format (CDF) parameters (based on Cadence's name for them), are used to store parameters like transistor width and length, model information to be used for simulation, and other electrical and geometric attributes of the device or cell they are associated with. The *CDFInterface* class

[6] The Python OpenAccess wrappers used in the BAG framework have recently been fully rewritten and integrated into a comprehensive tool called OaScript [33] that provides wrappers for the OpenAccess API in four common scripting languages: Python, TCL, Ruby, and Perl. Future versions of the BAG framework should switch to the new wrappers as they are better tested and much more likely to be supported over the long term. This should have no impact on the external interface of the OAInterface. class though so other classes that depend on OAInterface should not be affected.

provides a set of functions to read and write CDF parameters in BAG. For our implementation, because of our choice of the Cadence Virtuoso suite as the initial schematic entry tool, we implemented a subclass of *CDFInterface* to enable access to Cadence's CDF parameters through the SKILL language API. This class, *SkillCDFInterface*, depends on the *DBAccess* class for access to the Cadence process that is required to execute any SKILL code. In the future, a subclass of *CDFInterface* could be written to support the iPDK's TCL-based iCDF parameters or other proprietary CDF alternatives.

2.3.3.1.4 Technology

The *Technology* class provides the BAG framework access to process-specific information used during the generation of a circuit instance. This information is read in from a specially formatted YAML [34] file called *BAG_tech.yaml* which must be created for each new process added to the BAG framework. This file stores information about the allowable range for various device parameters as well as their default values. It also contains important information regarding simulation models and process corners.

2.3.3.2 Generator Description Classes

The hierarchical nature of circuit designs provides a natural structure on which to base the organization of the design framework. As shown in Figure 2.5, the *Module* class is the primary class containing a description of the circuit generator itself. The *Module* class is the basic building block used for storing and manipulating the design data. All circuits drawn at the schematic level have an associated class that inherits functionality from *Module*. The *Module* class provides functions for accessing and modifying the circuit's schematic hierarchy and sizing information as well as for linking the logical connectivity to one or more physical layouts (which will be discussed in greater detail in the next chapter).

The *Module* class defines an intermediate circuit representation format that contains the connectivity and parameter information for a specific cell. It has two main subclasses, *DesignModule* and *TestbenchModule*, which are respectively used for cells that contain representations of the circuits being designed and testbench circuits that provide the auxiliary circuitry needed to properly simulate the circuits being designed. *DesignModules* are connected hierarchically based on the structure of the circuit schematic hierarchy. *TestbenchModules* are linked to a specific *DesignModule* and are attached as child objects in the object hierarchy. Since the *DesignModule* is used as the device-under-test (DUT) within its associated *TestbenchModules*, it is usually desirable for the *TestbenchModule* to have a pointer to its parent *DesignModule* rather than to an independent instance of the *DesignModule*. That way, if the generator writer modifies the parameters of the parent *DesignModule*, they don't need to copy those parameters to all of the associated DUTs within the *DesignModule's* associated *TestbenchModules* before running simulations. The BAG framework attempts to automatically detect the presence of the parent *DesignModule* as the DUT within a specific *TestbenchModule* and create a pointer to it. This is illustrated in Figure 2.6, which depicts the object hierarchy of a comparator and two of its testbenches upon instantiation (a) and after automatic DUT detection (b).

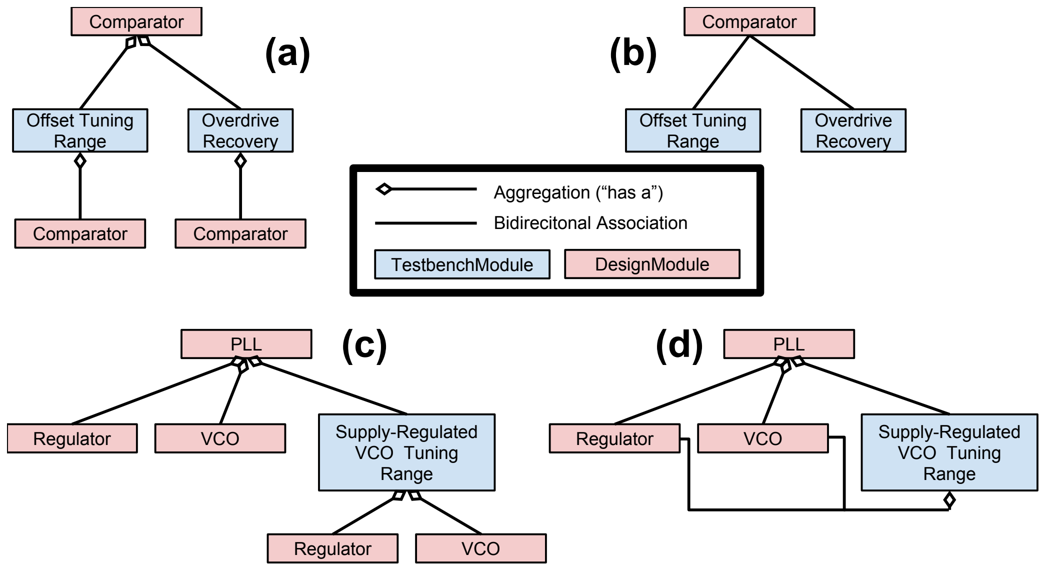


Figure 2.6: TestbenchModules with a single DUT before (a) and after (b) automatic DUT detection and with multiple DUTs before (c) and after (d) user-specified DUT detection.

Figure 2.6(c) shows a *TestbenchModule* used for characterizing the frequency tuning range of a supply-regulated voltage-controlled oscillator (VCO). Within the BAG framework, this *TestbenchModule* is associated with the PLL *DesignModule* rather than the VCO or regulator *DesignModules*. In general, *TestbenchModules* should be associated with the lowest level *DesignModule* that contains all of the DUTs used within that *TestbenchModule*. For simple testbench schematics that contain a single DUT, the link from the testbench DUT to the parent *DesignModule* is made automatically. For the PLL example shown in Figure 2.6(d), the link must be made manually in the initialization function for the PLL object.

Modules contain several attributes that store the schematic/netlist information for a circuit. *Instances* is a python dictionary in which the keys are the instance names of all the subcircuits of the current module and the values are the corresponding *Modules* associated with each instance name. The *Module* object also contains attributes that point directly to each of the instances listed in the *Instances* dictionary. This provides a shorter notation when accessing subcircuits, especially when they are several levels deep – e.g.

```
pll.loop_filter.amplifier.M1
```

instead of

```
pll.instances['loop_filter'].instances['amplifier'].instances['M1']
```

In addition to the subcircuit instance information, the connectivity for the cell is also stored in the *Module* object's child objects. *Pins* is a dictionary that stores the input/output connection points for the current module. *Connections* is a list of connectivity between nets and pins, both the pins at the current level of the hierarchy and the pins of the subcircuit instances. The net information from the *Connection* objects can also be accessed though a child object of *Module* called *nets.*

2.3.3.2.1 PyNetlist

The data structures for instances, pins, and connections stored in the *Module* are also grouped together in a class called *PyNetlist*. A *PyNetlist* object contains the minimal amount of netlist information to translate a *Module* instance to or from a schematic or netlist. The *PyNetlist* class can be initialized from a YAML file containing the netlist information that is created directly from a schematic template made in Cadence. It can also be created from an existing Module object and then parsed to create a netlist for use in simulation or LVS operations. A *PyNetlist* object is essentially a generator instance or a generator skeleton that respectively contains the circuit sizing and connectivity information or a placeholder for that information. Unlike the *Module* objects, it does not contain any information about the design procedure.

2.3.3.2.2 Parameters

Parameters are a key attribute shared by all types of *Modules*. They store the process-specific sizes associated with a generated circuit instance. Each parameter has several attributes including a name, value, default value, and an optional callback function that is called every time the parameter value is updated. All the parameters for a particular *Module* are stored together in a dictionary-like container object that is a direct child of the *Module* object. Pointers to each parameter are also created as children of the *Module* object for ease of access. Parameters are created automatically during the template schematic and template testbench schematic import process and can also be added by the generator writer in the initialization function of a specific *Module* subclass.

Setting and adjusting parameter values is one of the most common tasks a generator writer must perform, so the framework provides several ways of simplifying and speeding up parameter setting. One way is by overloading the assignment operator to simplify code for setting a single parameter value. If a parameter is assigned to a *Parameter* object, e.g.

```
amp.M1.width = Parameter('width',desired_width)
```

then the assignment operator functions as expected. If, however, the parameter is set to a variable that stores a number, e.g.

```
amp.M1.width =  desired_width
```

then the value of the existing parameter is assigned to the number in question. Without overloading the user would have to write

```
amp.M1.width.value = desired_width
```

Parameters can also be set en masse using the *set_parameters()* function at the Module level. By default, this function recursively sets all parameters with the specified name, and at the level of the current module and below, to a specific value. This is useful for setting default values for a module, such as channel length or multiplier, which can then be overridden on a device-by-device basis if necessary.

2.3.4 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 circuits, so it is useful to create helper functions that aid with the specific sizing and layout procedures. The BAG framework groups them into *helper classes* associated with specific circuit categories in order to enable reuse and to reduce the effort required for the initial codification of a generator. This section will describe *helper classes* for primitive devices, custom digital logic, and circuits with connectivity that depends on the input parameters. Layout related *helper classes*, which we refer to as layout styles*,* will be described further in the next chapter.

2.3.4.1 Primitive Devices Classes and LookupTable

BAG includes helper classes for primitive devices – i.e. transistors, capacitors, and resistors – that provide the generator writer access to device characterization data for the primitive device in any technology node. This allows designers to write sizing procedures in terms of technology independent function calls that reference 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. The helper classes also provide callback functions for the primitive device parameters that verify that the parameter values fall within the valid range and on the grid specified by the process. Figure 2.7 shows an example of a primitive device template for an NMOS transistor and several generated instances resulting from that device.

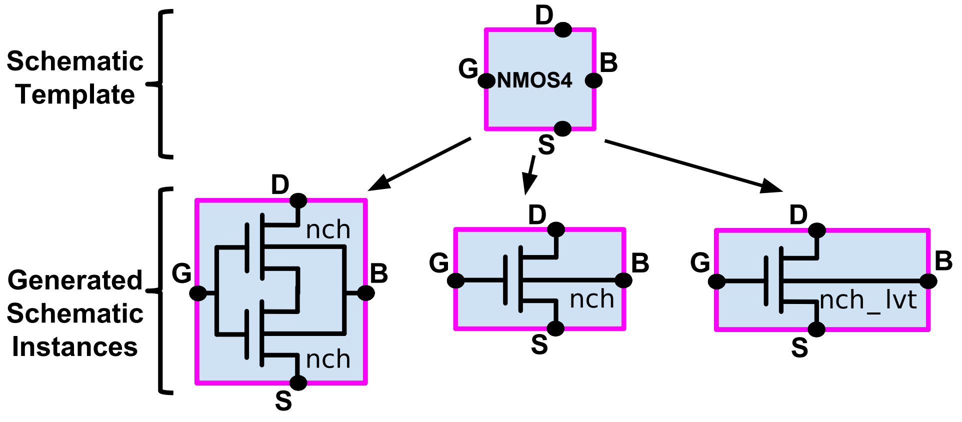
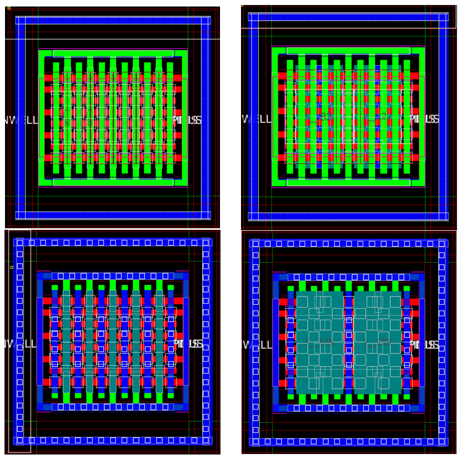
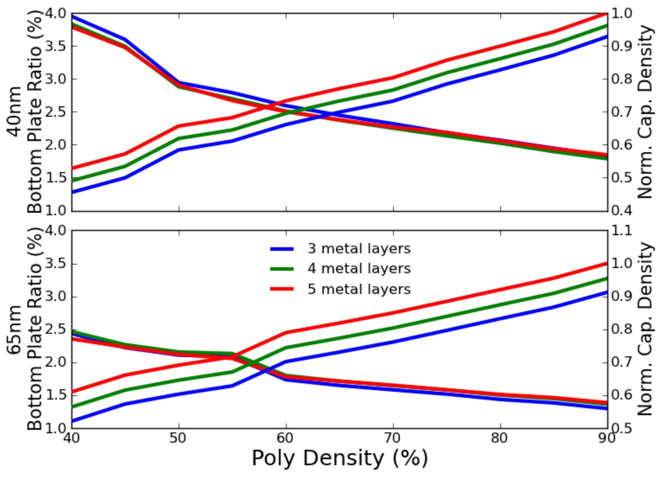


Figure 2.7: NMOS primitive device class schematic template and three example instances with different parameters.

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 [35], 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 primitive device 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 transistor helper classes include a *size_for()* function that allows the designer to set the device width based on the desired g_m/I_d. When the generator writer builds the template schematic for a particular generator, they build it out of primitive device templates. Then, they size the devices using the *size_for()* function without knowledge of the specific process technology parameters that will be used in the instances created by the generator. In Figure 2.7, the generated schematic instance sizes are created during the execution of the generator when the *size_for()* function looks up information from the target process' saved lookup table.

Another example of primitive device characterization is used in the SC DC-DC converter generator, presented in more detail later in Chapter 4. The performance of the converter is highly dependent on the capacitance density of the flying capacitor unit cell used in the design. The primitive helper classes are used to help characterize the capacitor density and bottom-plate ratio of the capacitor cell. In the plots of Figure 2.8(b), each point on the curves corresponds to a simulation – for 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 finger capacitor. The cell has a programmable size, aspect ratio, polysilicon density, and number of metal layers for the MOM capacitor, as shown in the four layouts of Figure 2.8(a). 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.





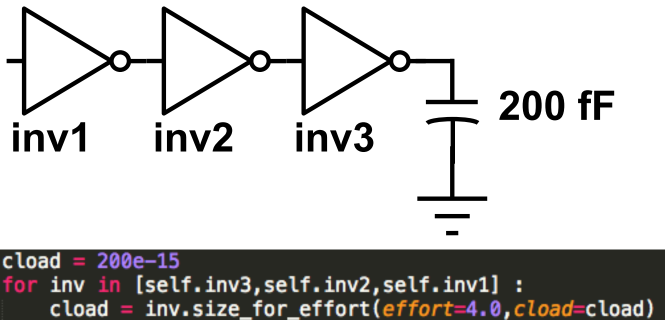
(a) (b)

Figure 2.8: Extracted capacitor density and bottom plate ratio simulation results (b) for a unit capacitor PyCell (a) in commercial CMOS 40nm and 65nm processes.

In order to help facilitate the creation and maintenance of the device characterization lookup tables associated with device primitives, the *LookupTable* class provides a generic interface for initializing and maintaining a table of extracted performance metrics. When a *LookupTable* object is first initialized, it runs a large set of simulations by sweeping a specified set of input parameters and saves the input parameter values and output performance metrics to a file. Functions in the primitive device helper classes, such as *size_for(),* then call a search function to locate the desired record in the lookup table. If the parameters of the lookup table do not meet the specified tolerances, then the *LookupTable* object can automatically launch further simulations in order to try to find points that do meet the specified tolerances.

2.3.4.2 Logical Effort

Most AMS circuits have some number of digital logic gates for control or driving purposes. The method of logical effort [36] is a common technique used to size a chain of logic gates in order to optimize the overall path delay given a target load capacitance. *LogicalEffort* is an abstract base class that provides access to necessary process information – through links to the *BAG_tech.yaml* file information stored in the *Technology* class – and defines a simple interface for custom digital logic gates to implement logical effort based sizing routines. For now, the class only requires each module that inherits from *LogicalEffort* to have a function, called *size_for_effort()*, that can size a logic gate for a specific target effort as well as load capacitance and can return the input capacitance. In the future, the class could be expanded to also handle calculations for the optimal effort for a given path. Figure 2.9 shows how to call the *size_for_effort()* function to size a chain of 3 inverters for a target effort of 4.0 and a load capacitance of 200 fF.



```
cload = 200e-15
for inv in [self.inv3,self.inv2,self.inv1] :
    cload = inv.size_for_effort(effort=4.0,cload=cload)
```

Figure 2.9: Sizing three inverter chain using *LogicalEffort's size_for_effort*() function.

2.3.4.3 Variable Structure Device

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. Figure 2.10(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 and trees, with an *ArrayModule* class that can procedurally create the circuit schematic structure using the base sub-elements of the structure and a simple specification of the pattern. *ArrayModule* inherits functionality from a more general *VariableStructureModule* class that provides the machinery necessary to adjust the module connectivity according to certain parameters. The code listing for an example array module can be found in Appendix A.1.1.

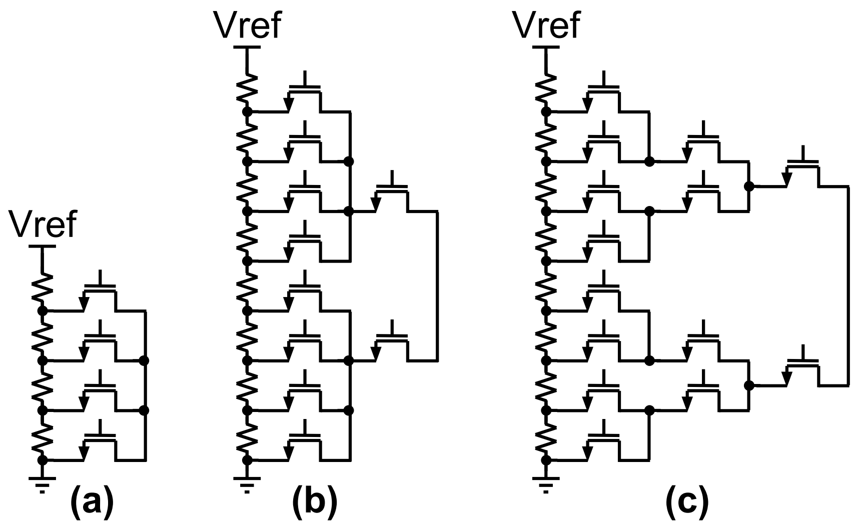


Figure 2.10: Three specific instances of a variable structure architecture for a resistor string DAC with bits = 2, radix = 4 (a), bits = 3, radix = 4 (b), and bits = 3, radix = 2 (c).

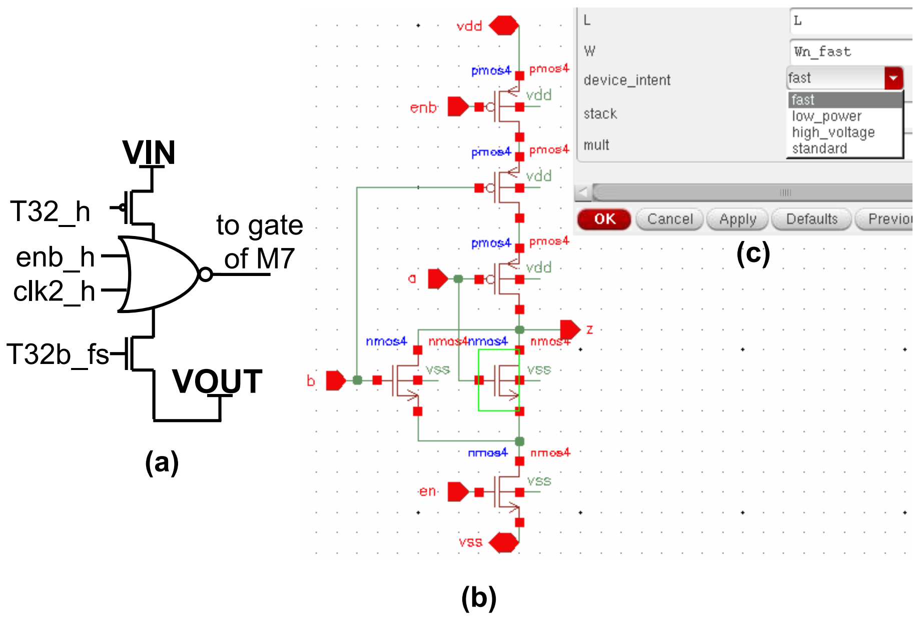
Each *VariableStructureModule* object has a set of regular *design parameters* and a set of parameters that can lead to a structural change called *structural parameters.* When a structural parameter is changed, it sets the value of an attribute of the module indicating that it requires a structural update. The designer can cause this structural update to happen immediately after the structural parameter value is changed by having the callback function for that structural parameter directly call the *update_structure()* function. By default, the *update_structure()* function is not called after every parameter change because that can waste a lot of time especially when the designer sets a large number of structural parameters in one contiguous code block. Instead of updating the structure after each one, it makes more sense to set all of the parameters and then just update the structure once. By default, operations that require an up-to-date netlist call *update_structure()* directly to make sure that the current circuit netlist matches the value specified by the structural parameters. As an example, the functions that launch simulations call *update_structure()* before doing any other action.

2.3.4.4 Veriloga

When cells have a variable structure, they often need a testbench with a variable structure. A simple example is a DAC; in order to measure the DNL of a DAC, we need a digital source with a variable number of output bits. One way to implement a source like this is to write some simple veriloga code. The veriloga helper class allows the designer to include veriloga blocks in their testbenches. Note that this class would also be useful in the future to allow simulations to swap out a spice model of a schematic cell with a veriloga model.

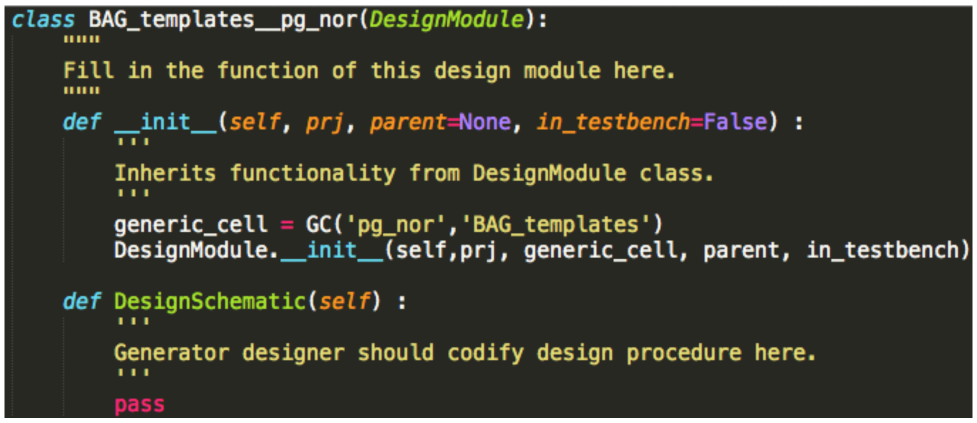
2.4 Design Flow Example

In this section, the design flow of Figure 2.2 is elucidated through an example design of the power-gated NOR driver shown in Figure 2.11(a). This design is relevant because it is used as a building block of the DC- DC converter that will be described in Chapter 4. Figure 2.11(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).



L: L
W: Wn_fast
device_intent: fast (fast, low_power, high_voltage, standard)
stack
mult

Figure 2.11: First design flow step of a power-gated NOR driver (a) illustrating parameterized schematic entry (b).



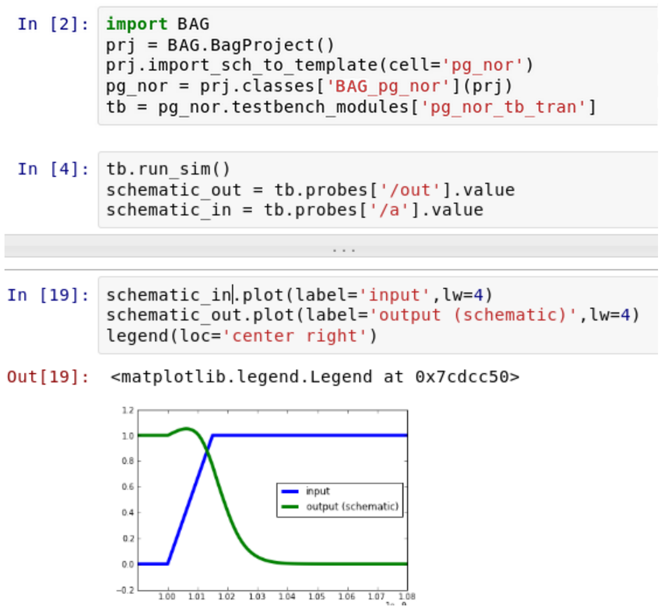
```
class BAG_templates__pg_nor(DesignModule):
    """
    Fill in the function of this design module here.
    """
    def __init__(self, prj, parent=None, in_testbench=False) :
        '''
        Inherits functionality from DesignModule class.
        '''
        generic_cell = GC('pg_nor','BAG_templates')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

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

Figure 2.12: Automatically created class stub of a power-gated NOR.

Next, the designer uses BAG to import the parametric schematic information and produce the stub class shown in Figure 2.12 as well as a stub class for a transient testbench. These classes enable the designer to instantiate a power-gated NOR object and its associated testbench in the interactive Python environment shown in Figure 2.13. From this environment, the designer can access the design hierarchy and parameters using a simple dot syntax – e.g.

```
pg_nor.NM1.W = desired_width
```

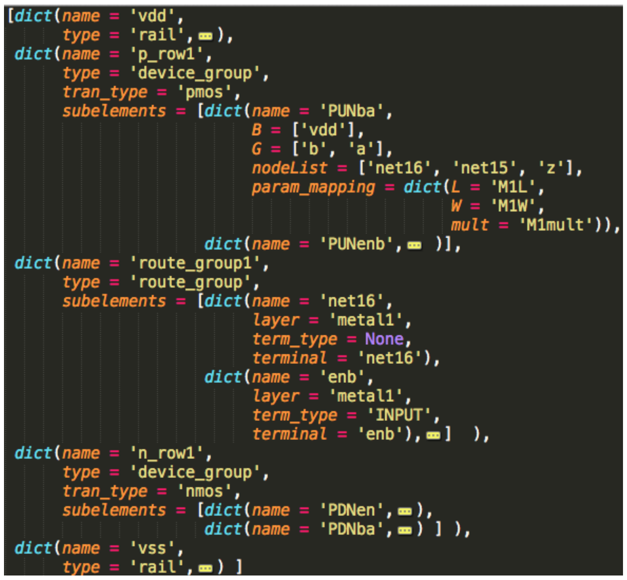
The designer can then – within the interactive environment – explore the design by choosing an initial sizing, simulating, viewing the results, and adjusting the schematic sizes based on the results.



```
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>
```

Figure 2.13: Interactive schematic exploration of a power-gated NOR.

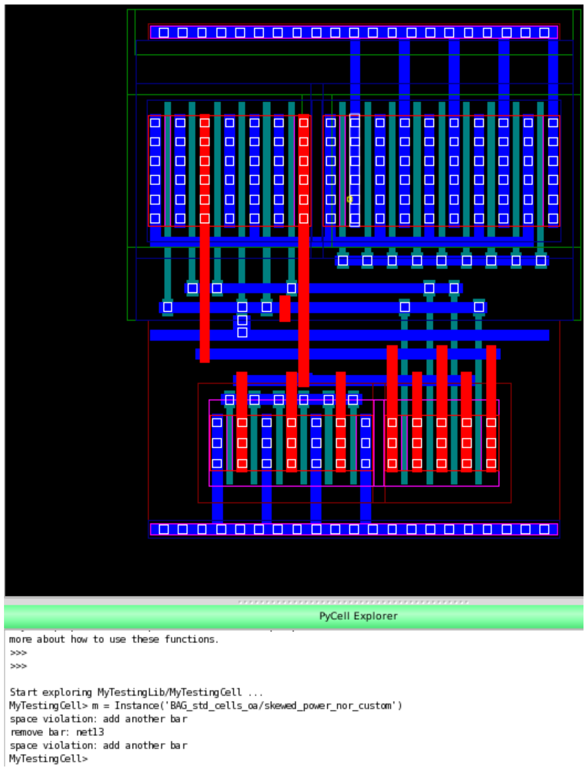
Within the BAG framework, PyCells for custom digital cells – like the skewed, power-gated NOR of this example– can be generated automatically. The netlist information is used to create an intermediate data structure – as shown in Figure 2.14 – that is readable by the Standard Cell layout style described later in Chapter 3 [37].



```
[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',...) ]
```

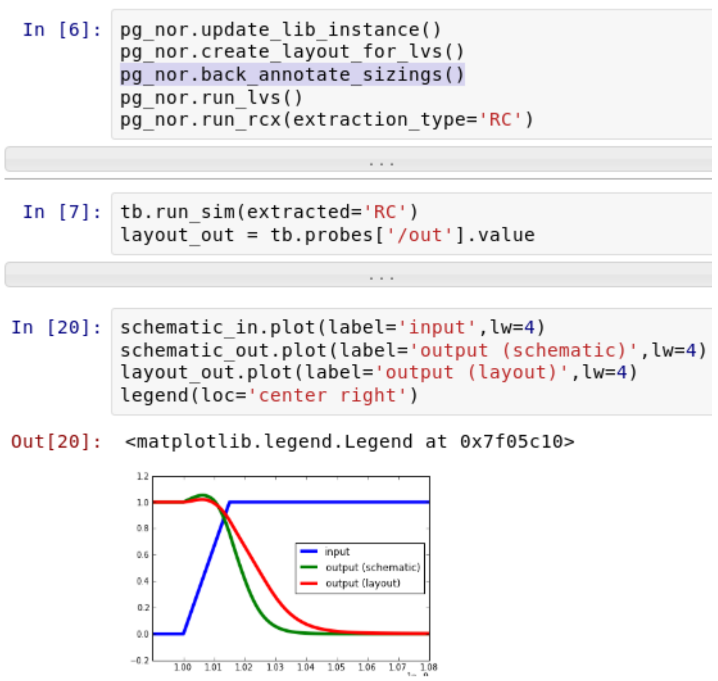
Figure 2.14 : Fully automated PyCell generation of a power gated NOR.

Figure 2.15 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 (Figure 2.16). The designer now has enough information to implement the methods of the generator interface into the BAG Gated-NOR generator class.



```
more about how to use these functions.
>>>
>>>

Start exploring MyTestingLib/MyTestingCell ...
MyTestingCell> m = Instance('BAG_std_cells_oa/skewed_power_nor_custom')
space violation: add another bar
remove bar: net13
space violation: add another bar
MyTestingCell>
```

Figure 2.15: Interactive layout exploration using PyCell Explorer.



```
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>
```

Figure 2.16: Interactive schematic exploration with post-layout extraction.

Chapter 3 – Layout

As mentioned in the previous chapter, a circuit generator framework must provide a way to codify all portions of the design process including layout. Unfortunately, analog layout generators are much more difficult to create than digital layout generators. In the digital world, layout generation is typically provided by place and route tools that rely on a functionally complete library of highly regular, easily-composable structures – i.e. digital standard cells – to automatically build hierarchical layouts from a circuit netlist. For analog circuits, the only functionally complete library of structures is the technology library itself. Since the devices in the technology library do not share a common template, they are not easily composable, and the tools for automatically placing and routing them are much more complex than the corresponding digital tools.

3.1 Layout in the BAG Framework

The layout portion of the BAG framework, just like the framework as a whole, must be simple to learn and easy to adapt for new layouts and new processes. Generated layouts should be LVS and DRC correct across most, if not all, of the allowable parameter space. If the designer encounters a corner case where the layout fails to meet DRC or LVS, it must be straightforward to modify the layout generator to fix the error.

Techniques for creating AMS layout generators generally fall into the same two categories as comprehensive circuit generators: optimization-based and knowledge-based. Optimization-based techniques usually require the circuit designer to enter constraints in a special format that is then passed to a tool that automatically places the devices and performs routing. In contrast to knowledge-based techniques, it is not necessary for the designer to write procedures that explicitly create geometric objects. There are several commercially available optimization-based tools for analog placement and routing including Synopsys Helix [38] and Pulsic's Unity Analog Router [39]. Due to limited time, we have so far focused on just a few knowledge-based techniques and have not included interfaces for these commercial tools yet. However, the intent of the BAG framework is to allow the designer to choose whatever method for creating parametric layouts they want, so interfacing with optimization-based AMS layout tools is on the development roadmap.

Layout generation efforts in the BAG framework have largely focused on knowledge-based techniques for AMS layout built around parameterized cells (Pcells). Pcells are procedures written by layout designers that read in user-specified parameter values and create corresponding geometric objects in the design database of the cell. When writing Pcells, designers must take care to account for all relevant design rules in order to ensure a DRC correct layout. Typically, these are written using Cadence's proprietary SKILL language, but the OpenAccess database provides a plugin interface so that they can be written in any language including C, TCL, or Python. Pcells are almost universally used for low-level device primitives – e.g. transistors – but avoided for even simple multi-device cells because of the amount of coding effort necessary to make a DRC clean layout.

In order to reduce this effort, Ciranova[7] introduced PyCells that still directly create geometry, but use functions that automatically check and account for the design rules so that the produced layout is DRC correct by construction. This helps designers create more robust layout generators with much simpler code; in fact, in many cases these generators can be directly compiled for several different processes without changing the code, as shown in Figure 3.1. PyCells do have some disadvantages though, for example large PyCells that do not take advantage of structural repetitions can have long compile times, an anathema to iterative design flows. The generator designer should thus carefully plan the demarcation of hierarchical boundaries and take advantage of the capability of caching the layout of sub-modules if possible.

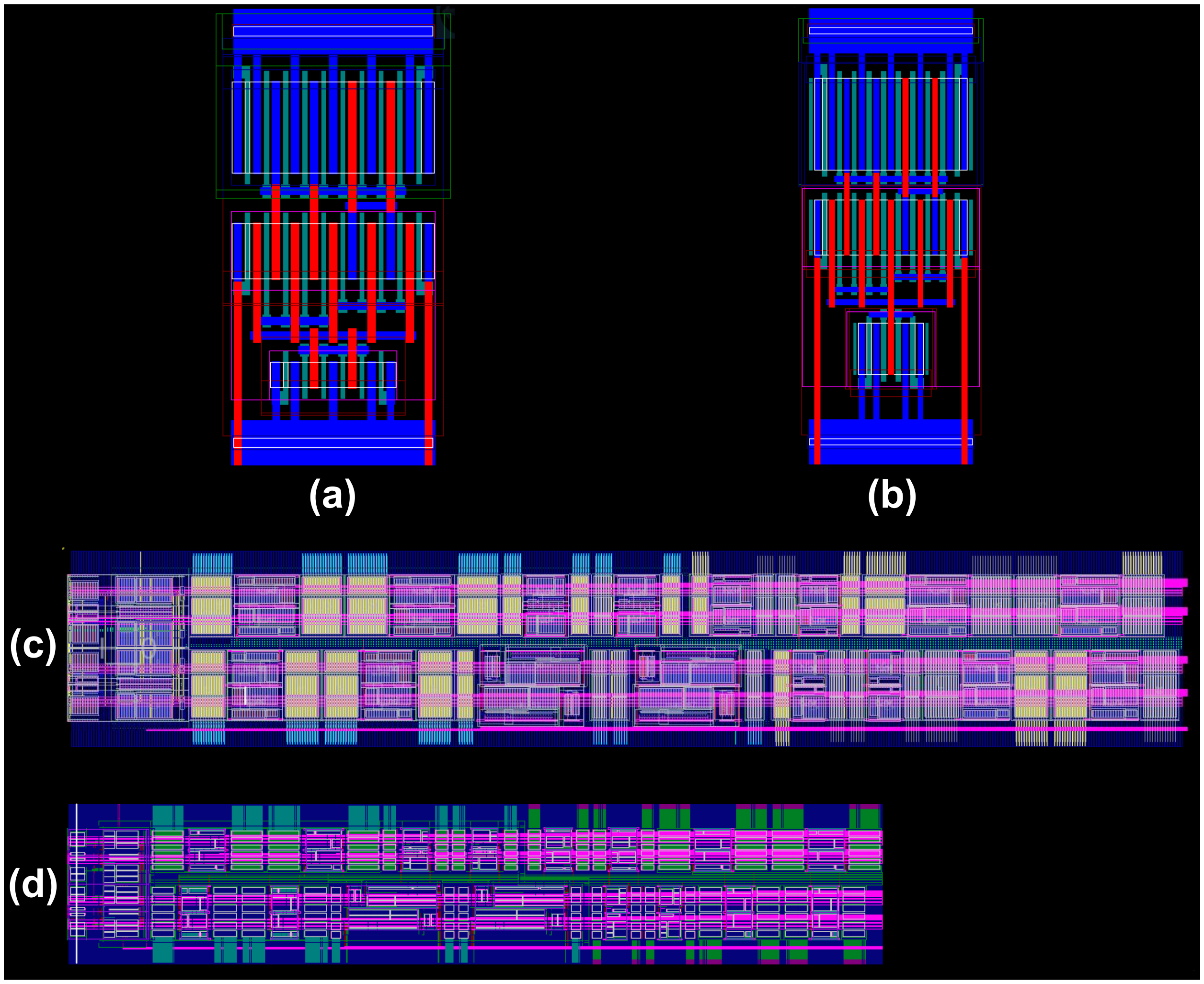


Figure 3.1: Instances of an amplifier PyCell compiled using a 65nm (a) and 40nm (b) technology file and of a network of power switches and drivers compiled using a 65nm (c) and a 40nm (d) technology file.

[7] Now Synopsys.

Figure 3.2 shows the three different paths enabled so far for creating layout generators in the BAG framework. All three rely on PyCells, but each requires a different amount of further design effort. For top level and high level cells with complex hierarchies, the designer can create a fully custom PyCell. The BAG framework provides some extra functions beyond the basic PyCell API, but of the three paths, this one certainly requires the most effort from the designer. At the other end of the spectrum, custom digital cell layouts can be fully generated as described in [37].

The contribution of this work is primarily for cells that fit in between these two categories. For AMS cells with only a few levels of hierarchy that fit into one of several categories of common layout styles, the BAG framework provides abstract base PyCells that the designer can use as a base for their PyCell code. These styles provide useful functions for performing common tasks in the layout and can even provide a complete layout procedure so that all the designer must do is enter parameters into a template data structure. Using only a handful of such layout styles, our goal is to provide a quick path to creating DRC/LVS correct layouts so that the designer can quickly begin design iterations including the effects of layout parasitics. Ideally, these layouts can be used with minimal modification, but in the event that they require modification, they are designed to be easy to alter and extend. The remainder of this chapter will focus on the four styles created for the BAG framework so far: *Standard Cell, Standard Row, Standard Block,* and *Array.*

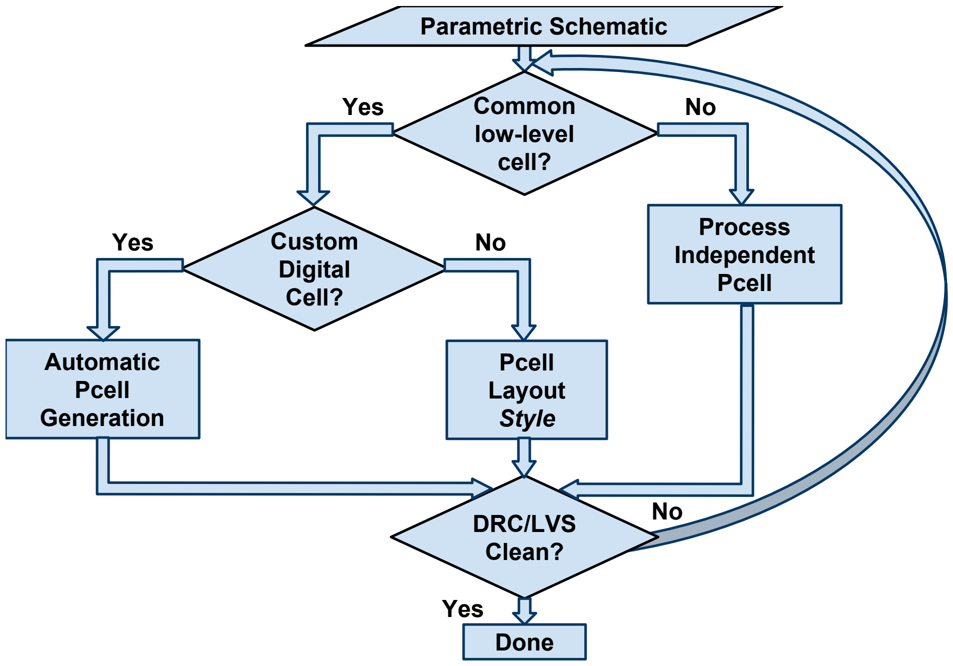


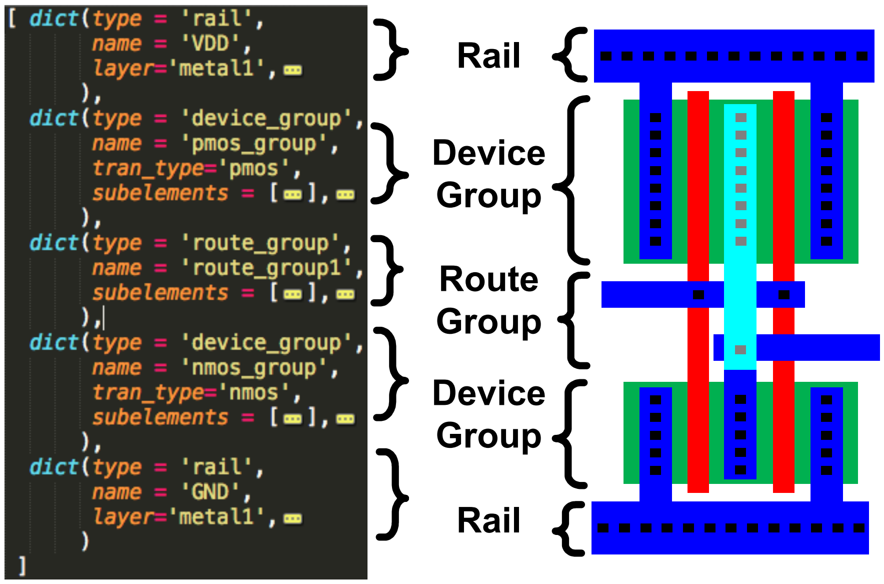
Figure 3.2: BAG layout generator flow.

3.2 BAG Standard Styles

The *Standard* family of layout styles – i.e. *Standard Cell, Standard Row,* and *Standard Block* – is intended for use as the default layout style for all AMS cells ranging from individual cells like amplifiers, comparators, and level shifters to larger blocks formed from hierarchical interconnections of cells. A general style that can handle almost any type of cell is very useful as an initial layout generator so that designers can easily create layouts early on in the generator writing process. This enables the designer to quickly identify and focus on the most important layout parasitics and use that information in the schematic sizing procedure. Beyond providing rapid layout feedback, the goal for the *Standard* styles is to create high quality layouts so that, for many cells, the initial layout is good enough (i.e. DRC and LVS clean with reasonable parasitics) – freeing the designer from spending any significant amount of time modifying the layout. *Standard* style cells also have programmable aspect ratios that make them ideally suited for placement around and between other AMS blocks with more rigid layout requirements.

3.2.1 Standard Cell

The basic unit of 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. 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 Cell* layouts, shown in Figure 3.3, consists of Python dictionaries that provide details about the layers and relative locations of power rails, transistor devices, and the 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 must extend or override the functionality of the *Standard Cell* class.



```
[ dict(type = 'rail',
        name = 'VDD',
        layer='metal1',⋯
      ),
  dict(type = 'device_group',
        name = 'pmos_group',
        tran_type='pmos',
        subelements = [⋯],⋯
      ),
  dict(type = 'route_group',
        name = 'route_group1',
        subelements = [⋯],⋯
      ),
  dict(type = 'device_group',
        name = 'nmos_group',
        tran_type='nmos',
        subelements = [⋯],⋯
      ),
  dict(type = 'rail',
        name = 'GND',
        layer='metal1',⋯
      )
]
```

Figure 3.3: ***Standard Cell*** **layout style.**

Standard Cell PyCells have a number of parameters that allow the designer to control e.g. 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, etc. Figure 3.4 illustrates several of these capabilities for an amplifier circuit; this figure also demonstrates an additional parameter that allows the class to automatically adjust the ratio of vertical space dedicated to NMOS vs. PMOS devices in order to equalize the number of PMOS fingers to the number of NMOS fingers and avoid wasting space. This is illustrated by the reduction of empty space next to the PMOS devices and the overall cell width between Figure 3.4(c) and Figure 3.4(d)-(f).

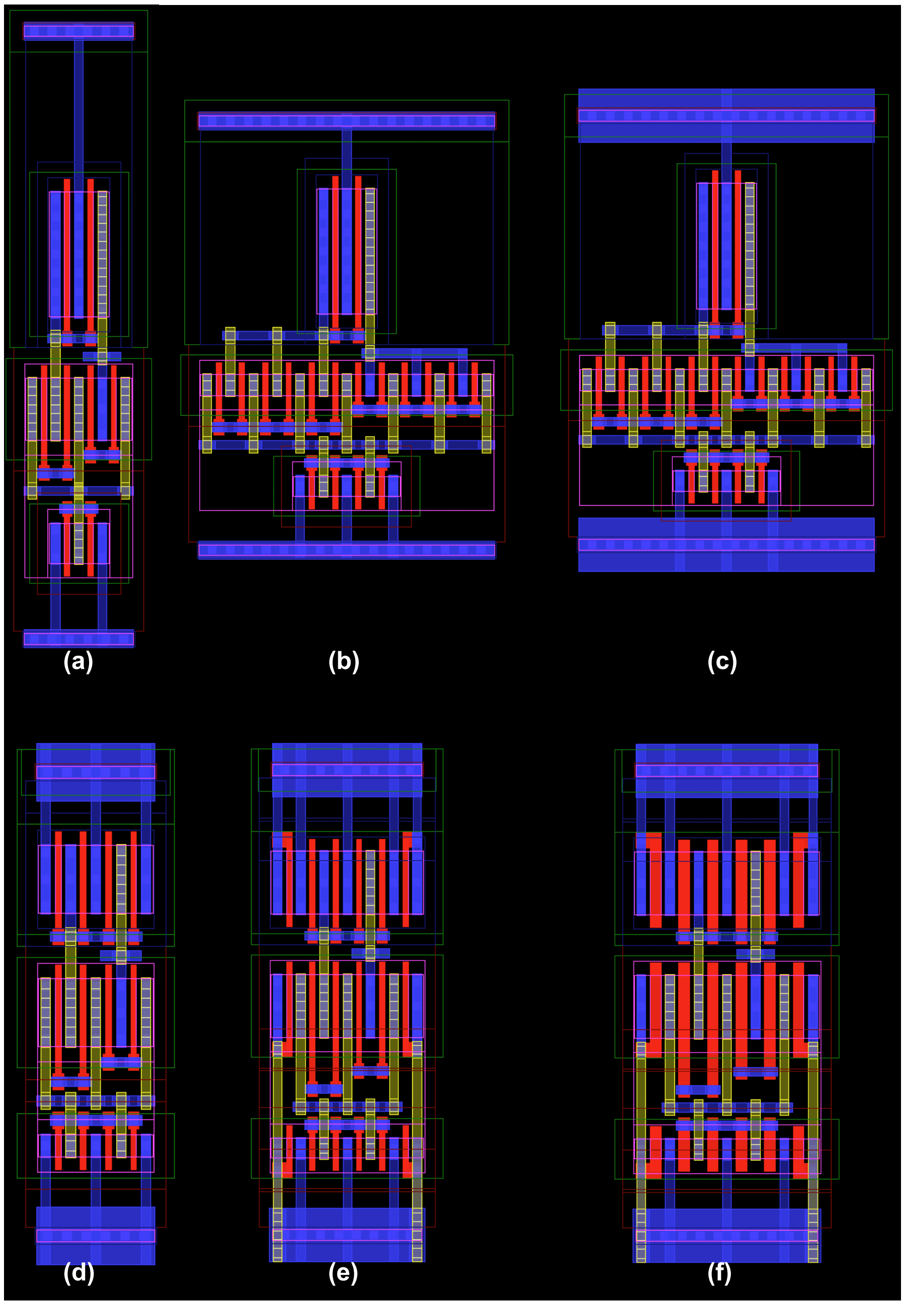


Figure 3.4: A 5-transistor amplifier PyCell (a) after sequentially lowering the vertical pitch (b), increasing the rail width (c), balancing the NMOS and PMOS ratios (d), adding dummies (e), and increasing the channel length (f).

The parameters available in *Standard Cell* PyCells are enabled by a python class that reads in the user-entered data structure and follows a specific procedure to generate a layout instance. This procedure, using the 5-transistor amplifier circuit shown in Figure 3.5 as an example, is as follows:

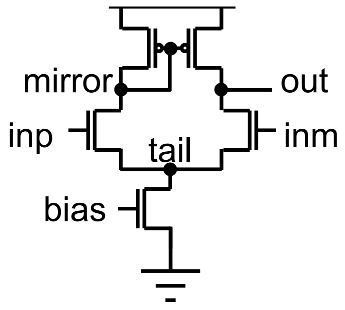


Figure 3.5: 5-transistor amplifier schematic.

1. **Initial Placement** – An initial placement of the three main components of all *Standard Cells* – rails, device groups, and route groups – is made using the order specified in the Python data structure. Placement is done relative to the *mason-dixon* line – i.e. the line through the middle of a routing group that separates the PMOS device groups (by default in the northern region of the cell) from the NMOS device groups. This initial placement, illustrated in Figure 3.6, makes use of the correct-by-construction PyCell methods to ensure a DRC-clean placement, but does not attempt to meet the vertical pitch constraint, which specifies the distance between the top and bottom rails.

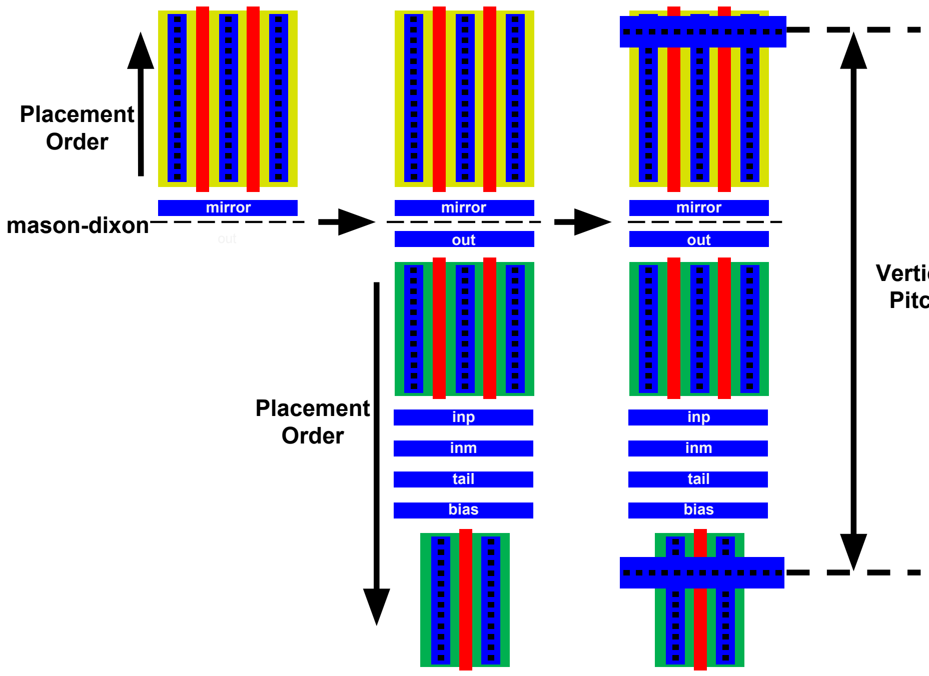


Figure 3.6: Initial placement of a 5-T amplifier in the *Standard Cell* style.

2. **Resizing Loop -** A device resizing loop iterates through the devices in each device group and attempts to fit them in the vertical space allocated for that device group (based on

ratios allocated for the PMOS to NMOS regions as well as each device group) by increasing the number of fingers in the device while maintaining a constant (subject to quantization error) device width. As shown in Figure 3.7, the loop completes when the devices have been resized sufficiently that the power rails can be placed such that they meet the cell's vertical pitch constraint.

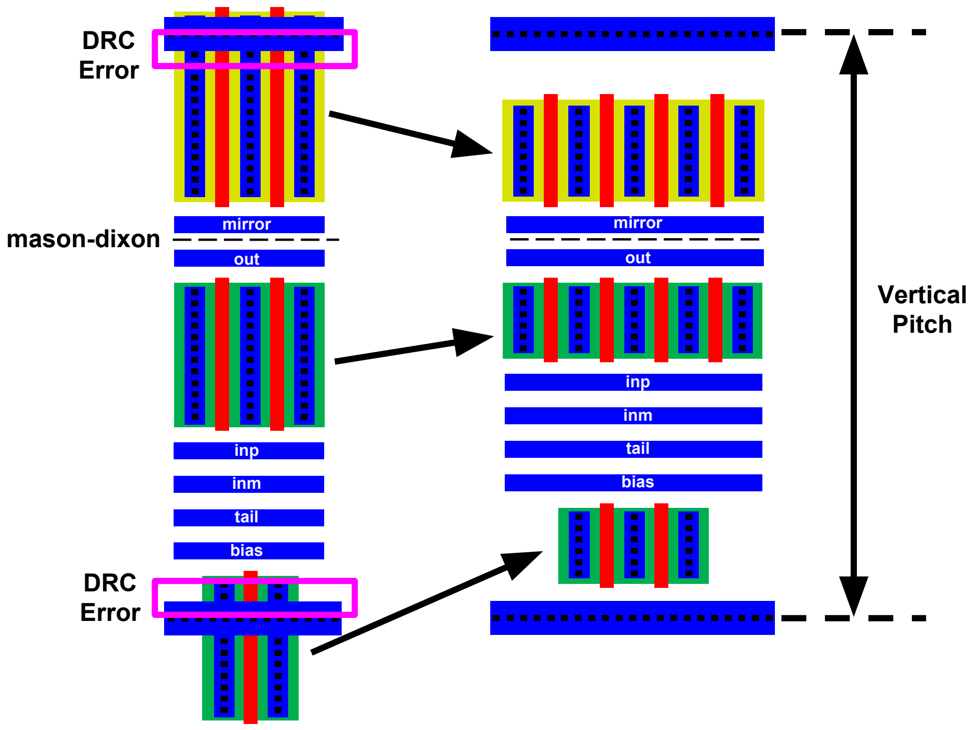


Figure 3.7: Automatic resizing of device groups in a standard cell in order to fit within the user-specified vertical pitch.

3. **Initial Routing -** Routing from the devices in each device group to the rails and to the routes in the route group is performed (see Figure 3.9(a)). The current routing mechanism relies on proper net ordering and device ordering in the data structure to avoid shorts. More sophisticated routing algorithms could be incorporated into the *Standard Cell* style in the future in order to simplify and/or improve the routing.
4. **Routing Repair** – The initial routing is not necessarily free of DRC errors so a cleanup step is performed that can repair certain types of DRC errors such as those created when devices from above and below connect routes to the same horizontal net and cause minimum spacing errors as shown in Figure 3.8(a). Drawing a rectangle to fill in the minimum spacing region can repair this. Also, if the contacts associated with the two routes are too close, the routing repair step simply removes one of the contacts. Another problem fixed during the repair step is associated with vertical routes from above and below connecting to adjacent horizontal nets, as shown in Figure 3.8(b). In order to repair this DRC error, the internal spacing between the horizontal nets is automatically increased, and steps 3 through 4 are rerun.

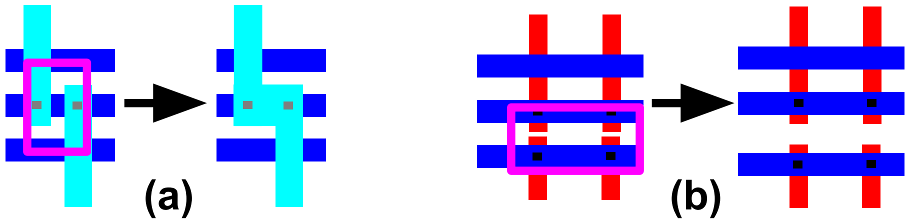


Figure 3.8: Two DRC errors and their associated fixes performed during the routing repair step.

5. **Routing Finalization** - The cell is finalized by trimming excess material from routes, creating pins, and adding/modifying enclosing layers as shown in Figure 3.9(b).

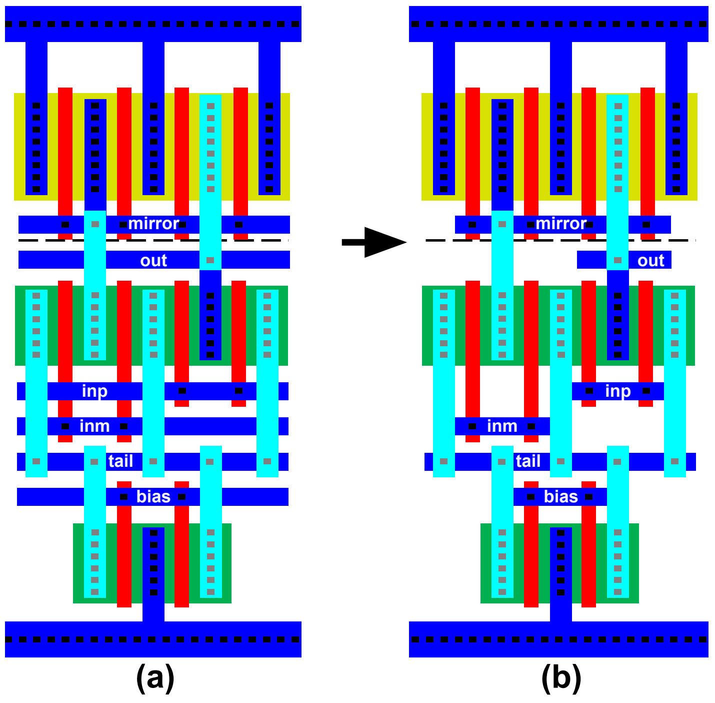


Figure 3.9: 5-transistor amplifier layout after initial routing (a) and routing finalization (b).

3.2.2 Standard Row

The power of the *Standard* family of layout styles is fully realized when several *Standard Cells* are connected together in such a way that the interconnected block still retains its programmability, especially the flexible aspect ratio. There are two styles in the BAG framework that enable the interconnection of *Standard Cell*s. The simpler of these, the *Standard Row* style shown in Figure 3.10, is used to connect a row of *Standard Cells* if their inputs and outputs are connected only to adjacent cells. The cells and the route groups used to connect them are placed following the order specified in a data structure that the layout designer creates in the PyCell file. An example PyCell that uses the *Standard Row* style can be found in Appendix A.2.2.

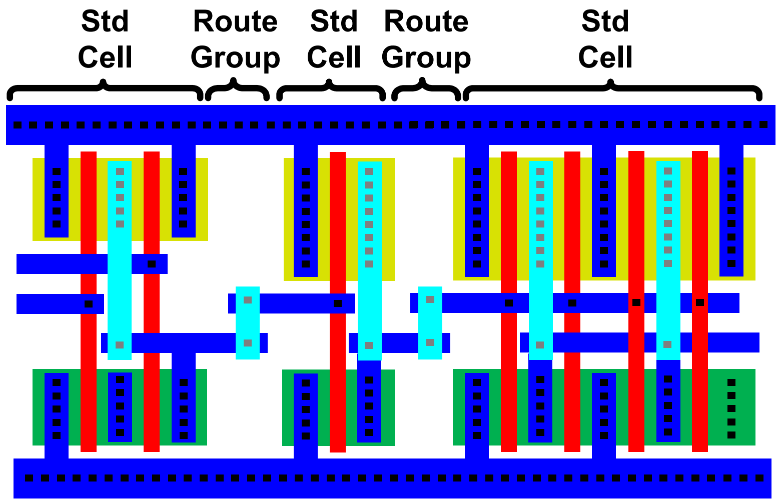


Figure 3.10: *Standard Row* style cell.

3.2.3 Standard Block

Standard Row is useful for small groups of *Standard Cells* with simple connectivity, but for more complex hierarchical blocks, fully specifying the routing can be challenging due to the limitations imposed by the *Standard Row* route groups and by the potentially large number of routes. Also, because *Standard Row* layouts are limited to a single row of cells, it can be difficult to enable a flexible aspect ratio when the number of cells in the row becomes large. To enable larger hierarchical blocks, the *Standard Block* style, illustrated in Figure 3.11, has the capability to stack rows of *Standard Cells* (or *Standard Row* objects), includes a more flexible semi-automated routing mechanism as well as a function for creating a simple power grid. An example code listing for a *Standard Block* cell can be found in Appendix A.2.4.

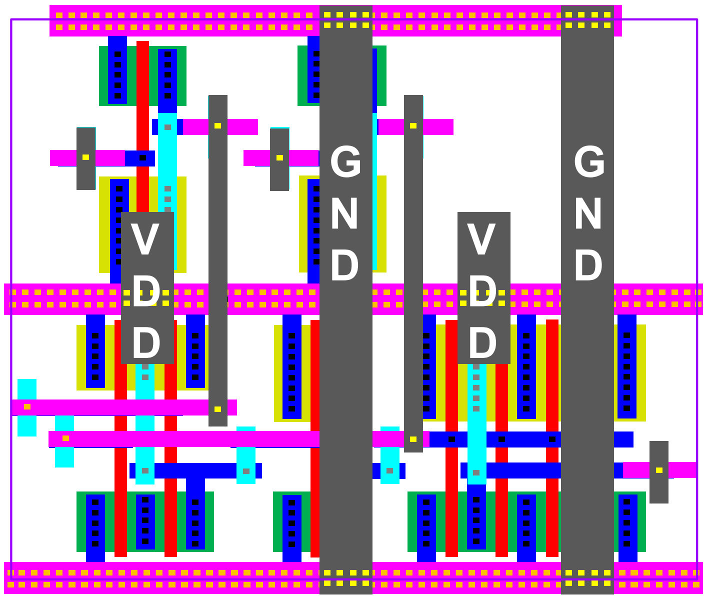


Figure 3.11 *Standard Block* style.

Standard Block style cells utilize the following algorithm to generate layouts:

1. **Initial Placement –** The initial placement of standard cells is made from left to right and top to bottom using the order specified in the data structure initialized in the PyCell code. By default, cells within the same row are placed next to each other as close as possible such that they still meet the DRC rules, and then rows are placed on top of each other and aligned to a center point. The layout designer can also specify explicit alignment points within two or more rows to control vertical alignment. Cells in different rows can be aligned by inserting extra space using the spacer cells shown in Figure 3.12(a). Figure 3.12(b) shows a spacer cell being inserted between two cells in order to avoid a DRC error.

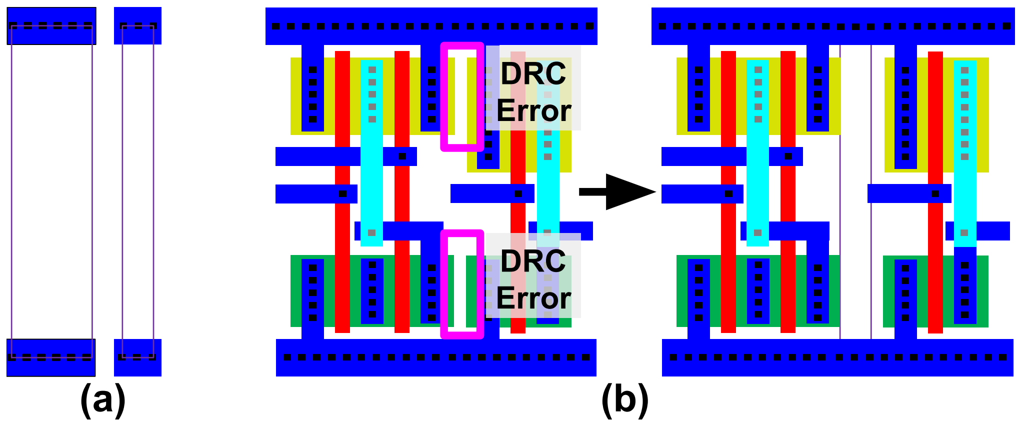


Figure 3.12: Two spacer cell instances (a) and the usage of spacer cells to prevent DRC errors during cell placement (b).

During the cell placement, the power rails of each cell are temporarily removed (see Figure 3.13(b)) so that placement operations only take into account the spacing between the cores of each cell and ignore the power rails, which will be connected together during the final step.

2. **Initial Routing** – Inter-cell routing works as follows:
   i. Input and output pins of *Standard Block* cells are placed vertically on metal4[8]. The designer can specify in the PyCell whether they want to allow the cell to automatically pick the location of these pins, or they can specify that they want a pin placed in a particular order within one of three bundles at the left, center, and right of the cell. Figure 3.13(c) shows a single routing channel being added to the rightmost bundle.

[8] The *Standard Block* style requires the use of the lowest four layers of metal inerconnect. It does not currently allow routing in higher metal layers but enabling this would not require major modifications to the code.

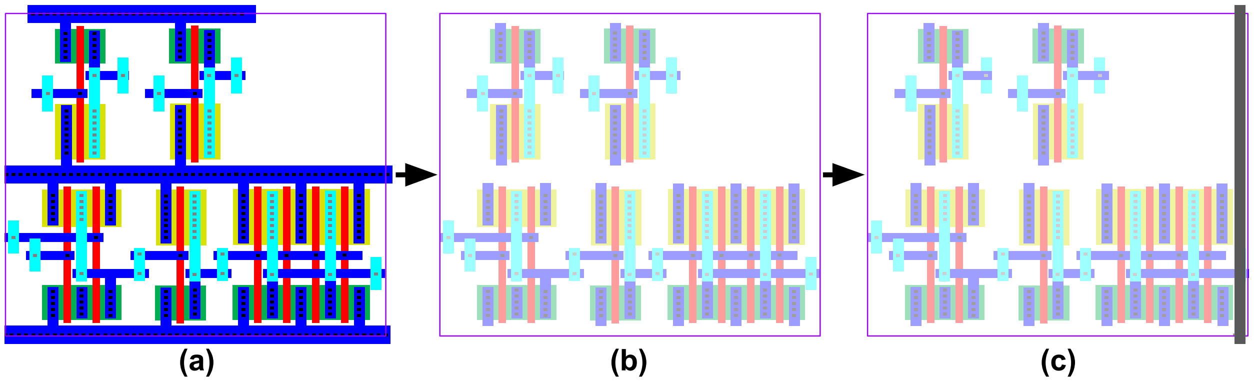


Figure 3.13: A *Standard Block* cell after initial placement with (a) and without (b) power rails and the same cell after the first routing step where a single vertical routing channel has been placed in the rightmost bundle.

ii. After the bundles are created, all the remaining nets that are connected to a top level pin are assigned to a free vertical routing channel from left to right in the order specified by the designer, as shown in Figure 3.14(a).

iii. Horizontal routing channels are assigned within each row for every net in the row (see Figure 3.14(b)). If the locations of the vertical routes associated with the horizontal route are known, then the horizontal route can be limited to a particular region, freeing up that horizontal routing channel for other routes. If not, by default, the entire length of the cell is temporarily dedicated to a single net.

iv. Vertical routing channels for any remaining nets that are shared between rows are placed on the x-axis as shown in Figure 3.14(c). The position of each vertical route is chosen so that the final length of interconnect required for the net will be minimized once the excess metal is trimmed from the horizontal routes in the cell finishing step.

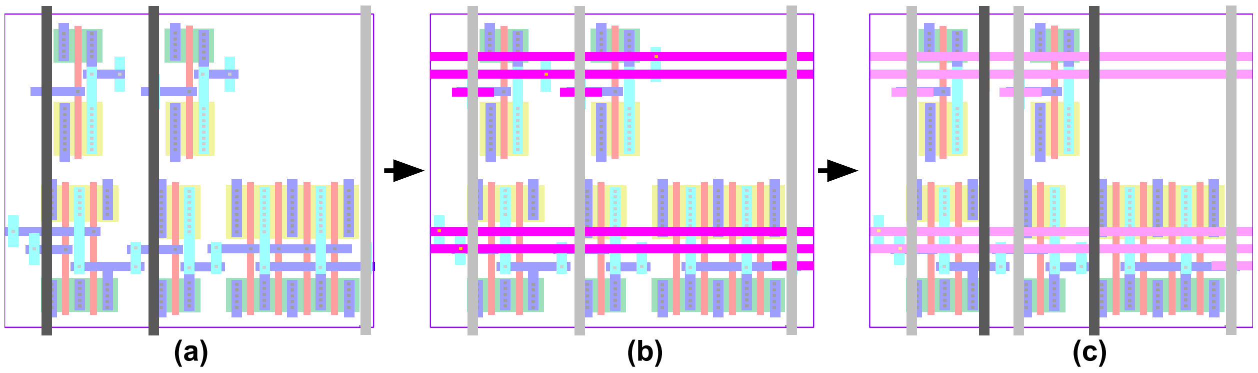


Figure 3.14: During the initial routing phase a *Standard Block* cell has all top level pins assigned to a vertical channel (a), has horizontal channels assigned to all intra-row nets (b), and then has a vertical channel assigned for any inter-row nets (c).

3. **Power Grid** – A simple power grid is created according to the power grid specification in the PyCell code. The specification sets a pattern for power rails and a target width and spacing. If the spacing cannot be met because of existing routing channels, the power rail is placed in the next available routing channel as shown in Figure 3.15(a). The power grid step can be run before the initial routing if the designer desires a more regular grid.

4. **Final Routing** – Given the location of all the vertical routes at the higher and lower metal levels, the horizontal routes can now be reorganized within the horizontal routing channels to minimize the length of each net and minimize the total number of horizontal routing channels required (see Figure 3.15(b)).
5. **Cell Finishing** – The power rails for each of the *Standard Cells*, removed during the initial placement, are added back in on a row-by-row basis and are strapped up to metal3 and connected to the power grid on metal4 (see Figure 3.15(c)). Excess material from routes and power grid rails is also removed in order to minimize parasitic coupling but it can be added back in by a higher level PyCell if it is needed for connectivity or to connect the power grid of two cells.

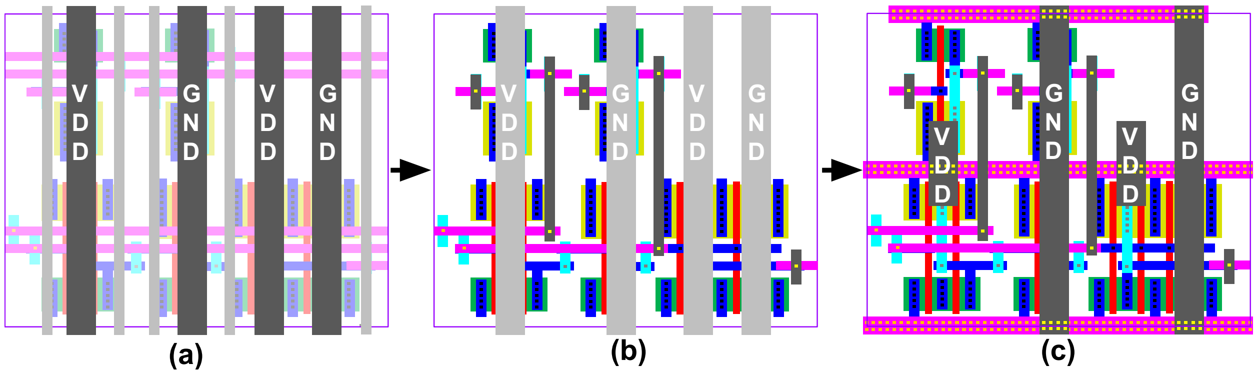


Figure 3.15: A *Standard Block* cell is finalized by adding a power grid (a), trimming and reorganizing routes (b), and rerouting the power rails on each row (c).

3.3 BAG Array Style

The repeated patterns that appear in the schematics of the variable tree structures described in 2.3.4.3 also appear in the layouts of those cells. BAG includes a layout *Array* style to simplify the creation of PyCells for cells constructed by repeatedly placing a handful of unit elements in a grid pattern. Figure 3.16 shows a floorplan of such a cell where an array of unit cells is connected using a common-centroid pattern. The style also includes provisions for inserting additional cells for auxiliary purposes - e.g., biasing and/or dummy cells.

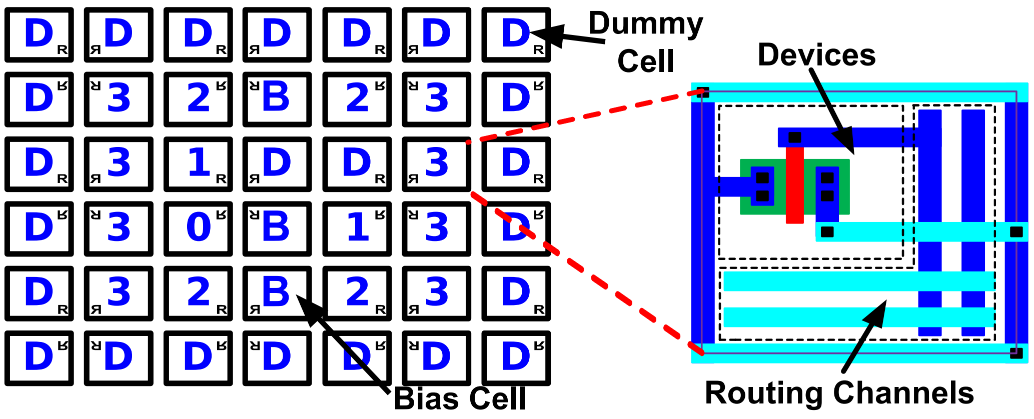


Figure 3.16: *Array* style floorplan of top level array and example layout of a unit cell.

Implementing a layout in the *Array* style involves two PyCells: the top-level array PyCell which should inherit from the *Array* style class, and the unit cell PyCell which can be a fully custom PyCell, or can inherit from another *style* class such as *Standard Cell*. The top-level array PyCell has parameters to adjust the number of cells (or number of bits) in the array, the unit cell placement pattern, the symmetry of adjacent cells, and the number as well as the arrangement of auxiliary cells. It also helps route control signals at the edges of the array.

The unit cell PyCell can use any style the designer wants, but must adhere to a few basic rules in order to function properly when it is instantiated by a top level array cell. First, it should be DRC correct when abutted on any side to other unit cells on its cell boundary. Also, as shown in Figure 3.17, it should have a parameter that can be used to control whether it is a regular unit cell, a dummy cell, or some other auxiliary cell like a bias cell. If it is difficult to write a single unit PyCell that can create all the different auxiliary cells, then separate PyCells for each type of auxiliary cell can be wrapped by a higher level PyCell which can then be used as the unit cell of an array.

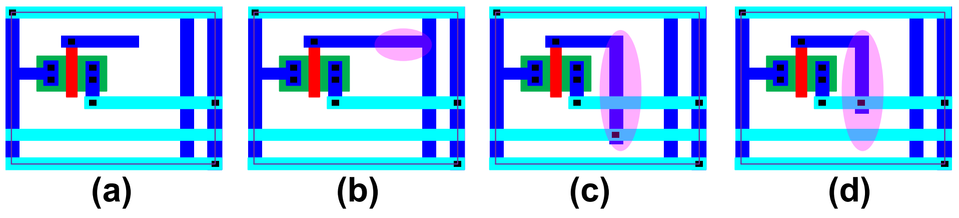


Figure 3.17: Using only the highlighted vias and stiches, a unit cell (a) can be configured for control by an odd bit (b), control by an even bit (c), or configured as a dummy cell (d).

Figure 3.18 shows three instances of a programmable array for two, three, and four bits of programmability. These instances use the default placement pattern of binary-weighted cells for *Array* Style cells. The main benefit of this pattern is that it only uses one horizontal and one vertical routing channel per control bit no matter how many bits are in the array. Even numbered control bits (as shown in Figure 3.17(c)) are routed to the left edge of the array horizontally and odd numbered control bits (as shown in Figure 3.17 (b)) are routed to the top vertically. If this pattern is not suitable for a particular application, the designer can select another placement pattern or can create their own.

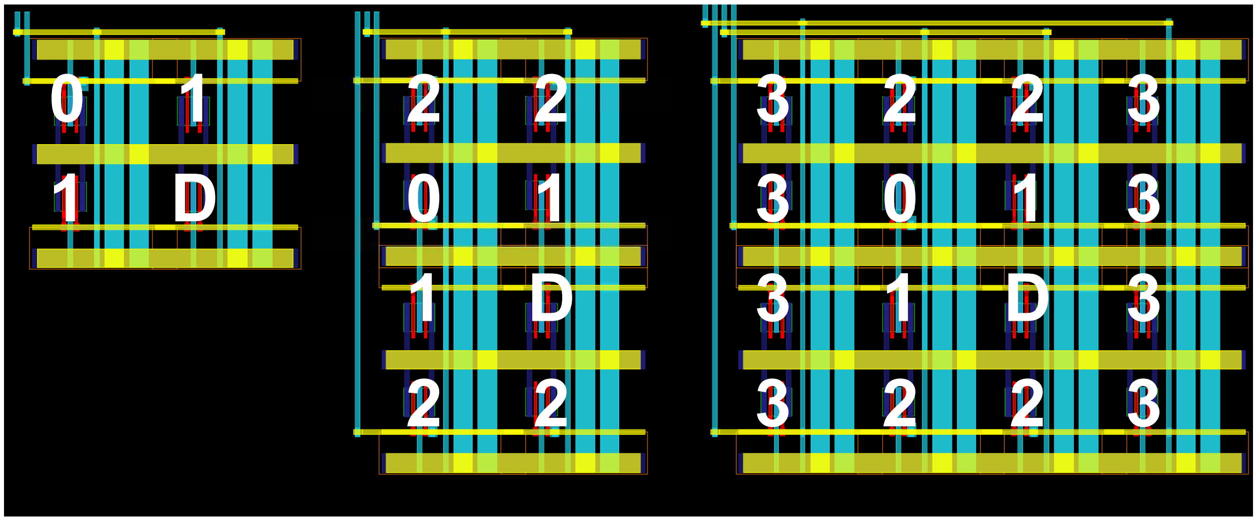


Figure 3.18: Three instances of binary weighted DACs 2, 3, and 4 bits.

3.4 Summary

An AMS circuit generator is not complete unless it can generate a full layout. There are commercially available tools for automating the creation of analog circuit layouts, but they have not yet seen widespread adoption by designers. We believe the main complaint designers have with these tools – most of which are optimization-based tools – is the difficulty in creating constraints that can help the tool perform a layout in a way that is similar to the way the designer would manually layout the circuit. Some readers might rightly point out that manual layout techniques can be non-optimal and might include unnecessary levels of symmetry or possess other aesthetically pleasing but electrically irrelevant properties. While this may be true and the layout produced by an optimization-based AMS layout tool might actually match – or even exceed – the performance of a manual layout, it is also true that it is much easier for a designer to debug a layout when they are familiar with it and understand the details of how it was created[9]. Constraint-driven optimization tools can often lead to non-intuitive results, leading to extra time spent adding extra constraints in order to force the tool to behave as the designer wishes.

Due to these issues with existing constraint-based approaches, the initial efforts towards enabling designers to easily create layout generators using the BAG framework have primarily focused on knowledge-based techniques that allow designers to fully specify the procedure for building a parameterized layout. Naturally, these layouts match more closely with the designer's plan but codifying a complex circuit can require many lines of code and a lot of time. To mitigate these issues, we created several Python classes that designers can use as the basis for building their own PyCell layout generators for a few common styles of AMS circuits. It is our

[9] It is also important to recognize that designers have an inherent bias to trust the layout they personally envisioned more than one created by something (or someone) else. This isn't a particularly justifiable or defensible bias but it is a reality that must be addressed.

hope that using just a handful of these layout styles, the BAG framework can provide a quick path for creating layout generators for 90% of the circuit generators created using the framework.

The main styles of the BAG framework are the *Standard* styles: *Standard Cell, Standard Row,* and *Standard Block.* These styles allow the designer to enter a design in a simple Python data structure wherein the order of elements in the data structure corresponds to the relative placement of the elements in the layout. They are also general enough to use for almost any circuit so they are useful for creating initial layout generators to provide accurate layout parasitic estimates early on in the design process. The *Standard* styles can be used for cells made of device primitives and/or groups of dissimilar cells; the *Array* style is used to create regular layouts for groups of a single unit cell. When combined with a unit cell layout built from the *Standard Cell* style, this can be used to build DAC and ADC structures and other regular arrays.

Chapter 4 – Integrated SC Regulator

The previous 2 chapters have described the functionality of the BAG framework using only simple example circuits such as basic digital gates and a 5-transistor amplifier. In order to demonstrate the true utility of an analog circuit automation framework, it must be used to create generators for more complex circuit blocks. The most complex circuit generator created using BAG so far is a fully-integrated Switched-Capacitor (SC) regulator. This chapter begins with some brief background on the operation of switched-capacitor converters before moving to a description of a fully-integrated switched-capacitor regulator architecture capable of responding to sub-nanosecond load current transients. The main topics of this chapter are a description of both the generator and the process of creating the generator for this switched-capacitor regulator architecture.

4.1 Switched-capacitor Converters

As the number of digital cores and peripheral blocks with separate voltage supplies in modern SoCs increases, integrated voltage regulators (IVRs) are becoming increasingly desirable as a means of providing multiple voltage supplies without requiring costly external components. Since a typical chip requires several independent supplies with different output voltage and efficiency requirements, the initial design effort spent creating an IVR generator can be immediately amortized across several instances. Furthermore, as IVRs are needed all over the SoC, having a layout with a programmable aspect ratio can help ease floor-planning by allowing the designer to fill in arbitrarily-shaped spaces on the die. IVRs do not typically have many critical, high speed input/output signals, so it is also useful to be able to adapt their shape to accommodate placement and routing constraints imposed by more critical high speed blocks such as transceivers or data paths.

Low-dropout regulators (LDOs) are a common choice for integrated voltage regulators, but their efficiency scales linearly with their conversion ratio (i.e. the ratio of the converter's output voltage to its input voltage), and in many cases this inefficiency is unacceptable. In order to achieve higher efficiencies, designers use switching regulators. The buck converter is probably the most popular step-down converter used for efficient power conversion, but it relies on high quality inductors, which are difficult to integrate on chip in a cost effective way. Switched-capacitor converters [40] rely on dense capacitors rather than high quality inductors, generally making them more well-suited to complete integration on the die.

4.1.1 Switched-Capacitor DC-DC Converter Operation

The basic operation of a simple two-to-one step-down – i.e. having an ideal voltage conversion ratio of ½ – switched-capacitor converter is depicted in Figure 4.1(a). Two non-overlapped clocks, Phase 1 and Phase 2, control a group of switches that alternately shift a capacitor, called the flying capacitor, from being connected between an input and an output voltage to being connected between the output voltage and ground. While this switching action delivers power from the input to the output at the desired conversion ratio, it also creates an undesirable ripple voltage shown on the V_{out} waveform at the bottom of Figure 4.1(a). The loss in system efficiency associated with this ripple can be improved by interleaving several of the switched-capacitor units. Figure 4.1(b) illustrates interleaving with two flying capacitors 90 degrees out of phase with each other. The number of interleaved phases along with the 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. For a more thorough analysis of switched-capacitor DC-DC converters, see [41].

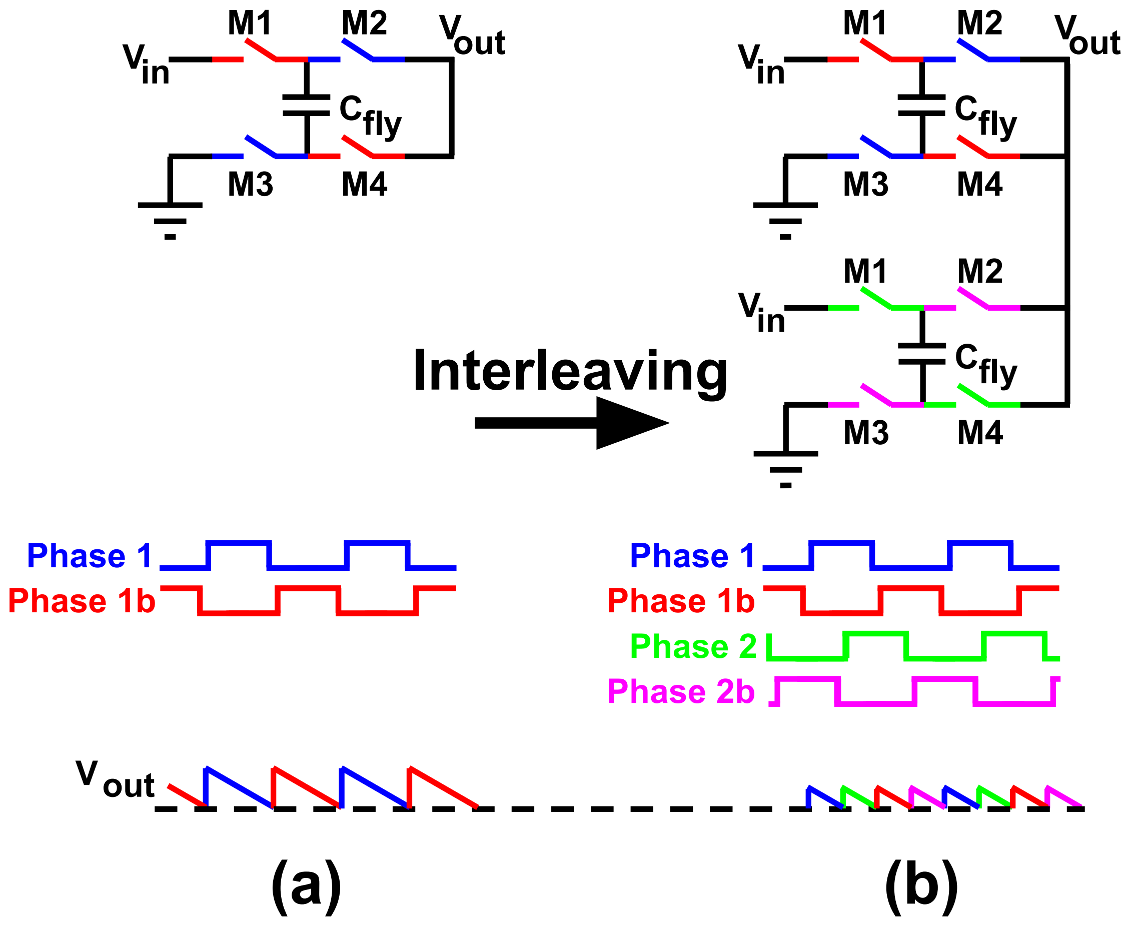


Figure 4.1: Switched-capacitor Operation.

When controlled by a fixed frequency, as in Figure 4.1, a switched-capacitor converter does not truly regulate the output voltage. In order to create a regulated switched-capacitor converter (i.e., a switched-capacitor regulator), the frequency of the flying capacitor switching action must be controlled through a closed loop that compares the output voltage to a desired reference voltage. One way of doing this is by using a lower-bound hysteretic feedback controller shown in Figure 4.2 [42]. The lower-bound comparator detects when the output voltage drops below the reference and triggers a flip in the state of the switched-capacitor switch network. This action regulates the output voltage to the desired value despite changes in input voltage or load current such as the current steps in Figure 4.2(c).

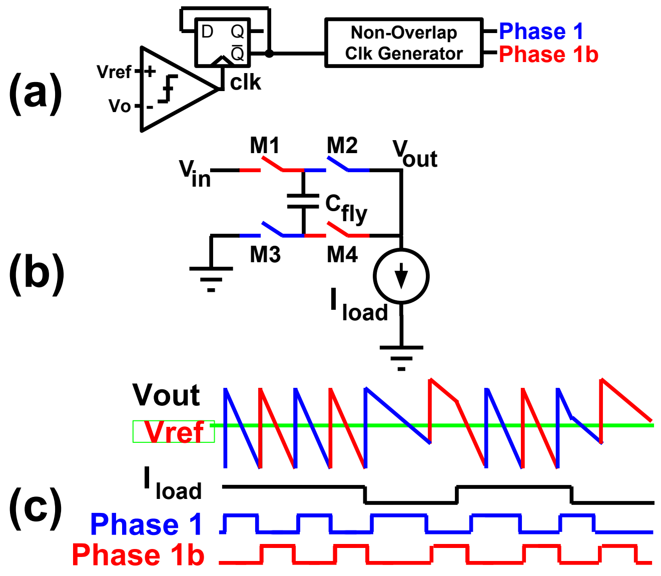


Figure 4.2: A lower bound hysteretic control circuit (a) used to control a switched-capacitor converter unit cell (b) and the waveforms (c) depicting its operation.

4.1.2 Switched-Capacitor Regulator Architecture

Since fully integrated SC converters are physically close to their loads and have fewer inductive and capacitive parasitics to deal with, they can inherently be made to respond faster than off-chip converters. If an integrated convert's regulation loop can fully capitalize on this capability, it can be used to enable fast dynamic voltage and frequency scaling (DVFS) which can lead to substantial power savings [43]. For this reason, creating regulation schemes that achieve very fast response times is an open and worthwhile challenge. Therefore, we chose to implement (using BAG) a generator for a switched-capacitor regulator that extends the work of [40]–[42] with a very fast response time that can maintain output regulation in the presence of sub-nanosecond load current transients.

SC converters have an inherent voltage conversion ratio determined by their topology. Unfortunately, as a regulation loop adjusts the output voltage and the effective conversion ratio shifts below the inherent ratio of the topology, the efficiency of the converter drops linearly. In order to mitigate this, the architecture chosen for the SC regulator generator uses a reconfigurable switched-capacitor unit cell [41] that can switch between 1/2 to 2/3 conversion ratios. This allows the converter to be reconfigured for higher efficiency depending on the desired conversion ratio. The regulation loop used to control the reconfigurable unit cells is similar to the lower bound control shown in the previous section, but modified to handle multi-phase interleaving and to maintain a loop response time that is as fast as possible. Each phase has a separate asynchronous comparator that triggers a flip in the state of that phases' switched-capacitor unit cell as soon as the output voltage drops below the desired reference. Once the output voltage has increased as a result of that flip in the switched-capacitor unit cell's state, the next comparator in the chain is enabled.

By having a comparator located locally within each phase, the critical path from the comparator to its associated switched-capacitor unit cell is minimized. This is crucial to maintaining a fast response to transient load current steps because the maximum switching frequency of the converter, and therefore the maximum load current that the converter can supply while still maintaining output voltage regulation, is limited by this critical path delay.

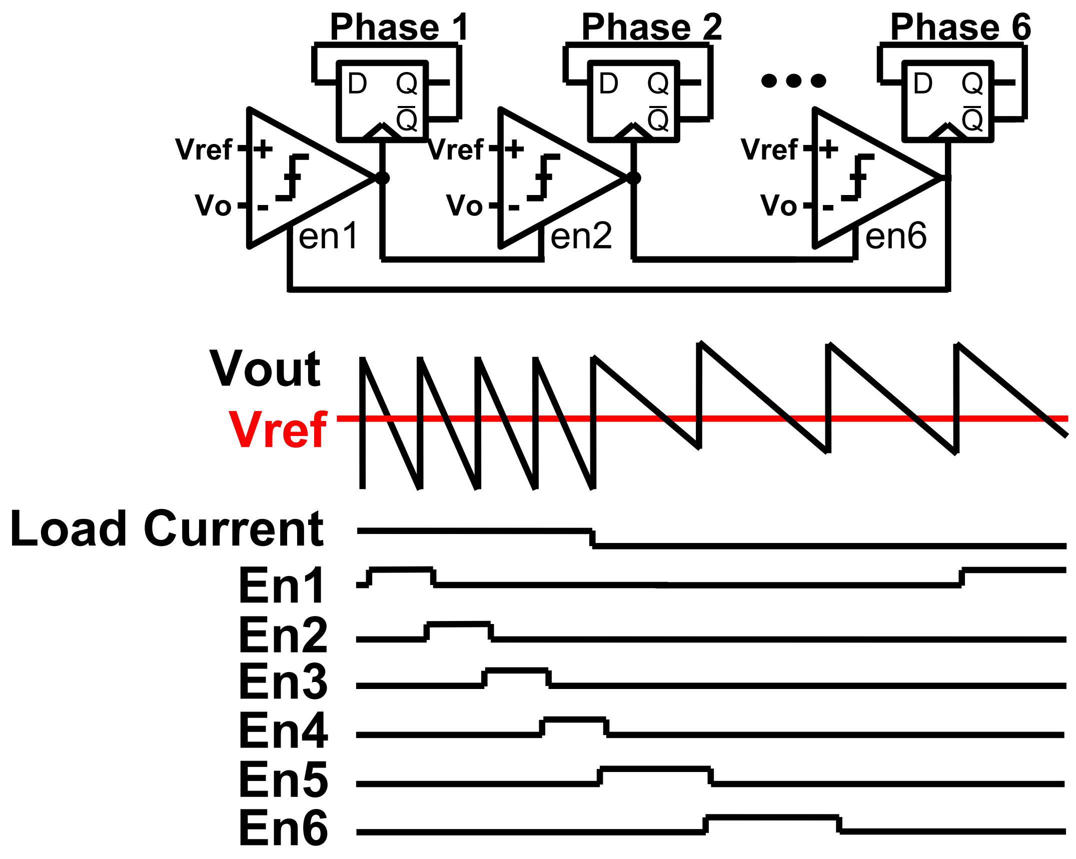


Figure 4.3: Implemented SC Converter regulation loop with asynchronous lower bound control and its behavior in the presence of a load current step.

4.2 Switched-Capacitor Regulator Generator

As highlighted in Figure 4.4, an SC regulator instance is made of an even number of interleaved phases (the exact number is determined at optimization time) and a corresponding number of bias current DACs. Each of the interleaved phases is itself made of four parts: a collection of flying capacitors, a network of power switches and drivers, a control core, and a load current DAC.

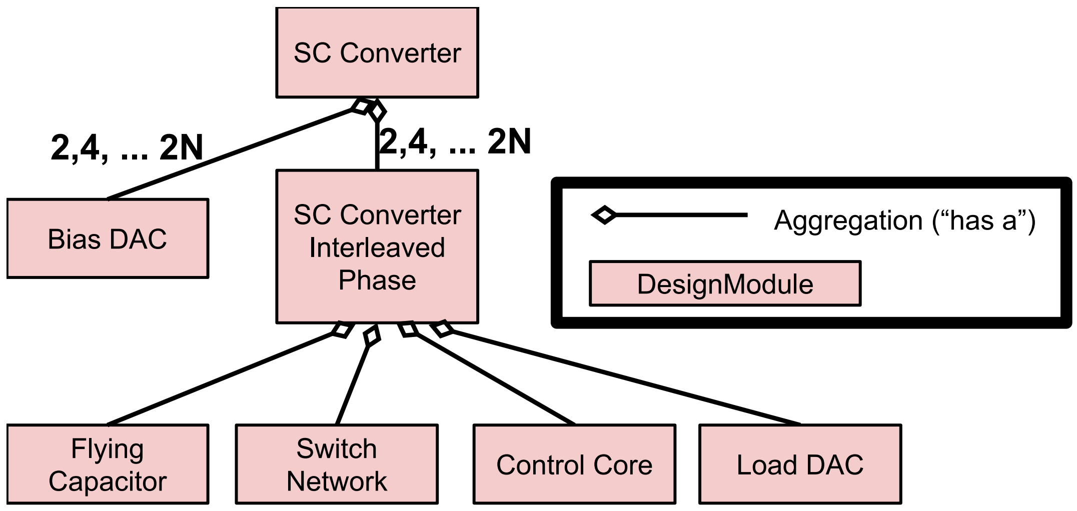


Figure 4.4: Switched-capacitor Regulator Generator.

In order to fully validate the functionality and flexibility of the BAG framework and to verify the quality of the resulting circuits, we utilized the SC regulator generator to create three different instantiations, each targeting different specifications in terms of total output power, power density, and aspect ratio, as shown in Table 4.1. The following section will describe the creation of a BAG generator for the regulator as well as the creation of generators for the main sub-blocks utilized by the regulator.

Specification	Regulator A	Regulator B	Regulator C
Power Density	$1 \, {}^{W}\!/_{mm^2}$	$0.2 \, {}^{W}\!/_{mm^2}$	$1 \, {}^{W}\!/_{mm^2}$
Output Power	$1\ W$	$100\ mW$	$100\ mW$
Aspect Ratio [H/W]	0.35	1.2	0.5

Table 4.1: Switched-capacitor Regulator Target Specifications.

4.2.1 Sub-Circuit Generators

We created generators for each sub-block in a bottom-up fashion so that during the development of the higher-level generators, the lower level generators would be available to instantiate the sub-blocks. However, during the execution of a generator, the specifications are typically passed top-down to each sub-block in order to optimize global performance – e.g. to create as compact of a layout as possible.

4.2.1.1 SC Interleaved Phase Generator

Each interleaved phase is made of two out-of-phase reconfigurable switched-capacitor circuits consisting of two flying capacitors and nine switches along with their drivers. Each phase also has a control block that creates the signals used as inputs to the switch drivers. The generator code for sizing this circuit is trivial because the higher level SC regulator generator handles its sizing. The layout generator is a fully custom PyCell and does not use a specific layout style. The PyCell’s main actions are wrapping the capacitor around the edges of the control block and switch network, creating routing channels between the capacitors to carry static control signals and the comparator enable signal, and driving the switch network and control block PyCells to fit in the targeted space.

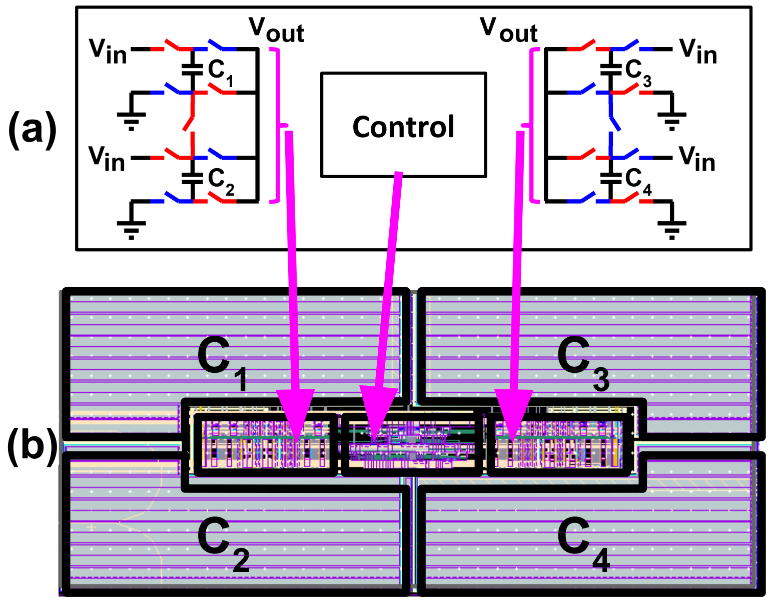


Figure 4.5: SC Interleaved Phase schematic (a) and layout (b).

4.2.1.2 Control Core Generator

The control core circuit's main function is to decide when to switch the state of the switched-capacitor unit cell. To that end, it contains an asynchronous comparator whose output drives a toggle flop followed by a phase splitter that outputs two 180° out-of-phase signals. These out-of-phase signals are then fed to a circuit that creates non-overlapping versions in two voltage domains – between V_{in} and V_{out} and between V_{out} and ground – in order to supply the control signals to the power switch driver circuits in the SC interleaved phase unit cell. The delay from the enable signal of the comparator until the time at which the output voltage is boosted from the charge added in by the switched-capacitor unit represents the fastest speed the converter can maintain regulation. Minimizing the delay of the critical path shown in Figure 4.6 is equivalent to maximizing the allowable switching frequency of the converter and is directly related to the achievable output power. This delay also limits the speed that the regulation loop can respond to changes in the output load.

The primary goal of the control core generator is to minimize the critical path delay. Sizing is done using the logical effort helper class (see 2.3.4.2) to size the elements of the critical path. First, the asynchronous comparator is sized for minimal power and a specified minimum delay. Then, using the driver size specifications passed down from the SC interleaved cell generator, the optimal fanout target is calculated and used to size the rest of the critical path. The falling edge detector is sized to prevent a race condition that can occur if a comparator is enabled before the output voltage has jumped due to the signal from a previous comparators output. In addition to the critical path, the control core generator also sizes measurement circuits that can be used to characterize each interleaved phase, circuitry for resetting the asynchronous comparator once the next phase's comparator has been enabled, and circuitry for modifying the effective reference voltage depending on the output load.

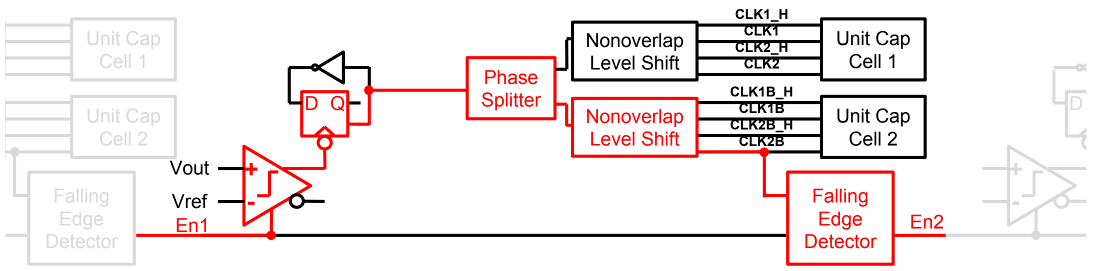


Figure 4.6: Control core critical path.

The generator of the control core layout inherits from the *Standard Block* class (see 3.2.3), and all of the subcircuits used in the control core inherit from either *Standard Row* or *Standard Cell*. Figure 4.7 shows asynchronous comparator instances implemented using *Standard Row*. These layouts use a feature of the *Standard* styles that allows the designer to create isolated power islands using deep nwell regions and breaks in the power rails.

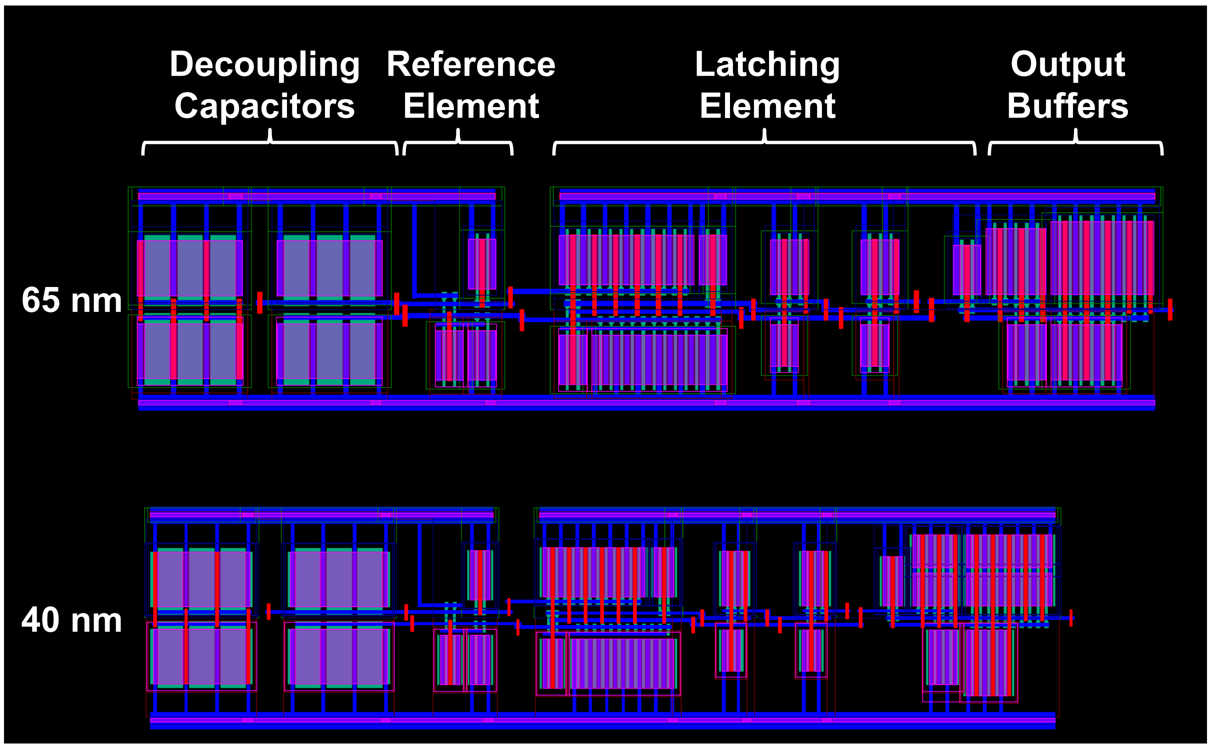


Figure 4.7: Asynchronous comparator PyCell compiled with 65nm and 40nm technology files.

An instance of the layout generator for the full implemented control core can be seen in Figure 4.8. The asynchronous comparator is highlighted in the bottom left of the figure to provide a sense of scale. The top three rows contain the non-overlap and level shift circuitry for the two phases of unit switched-capacitor cells. Their layouts are roughly mirror symmetric about the center of the y-axis in order to avoid mismatch. Control signals that interface with one SC cell or the other are placed in the left and right routing bundles while signals used for the asynchronous comparator and the shared portion of the control path are placed in the central routing bundle. The vertical portion of the power grid (thick yellow routes) is placed after the routing with extra spacing allocated to avoid adding extra load capacitance to any high speed signals. The power grid rails are also trimmed to avoid extra load capacitance, but this can be disabled with a parameter if a higher level PyCell needs the power grid to reach the borders of the cell.

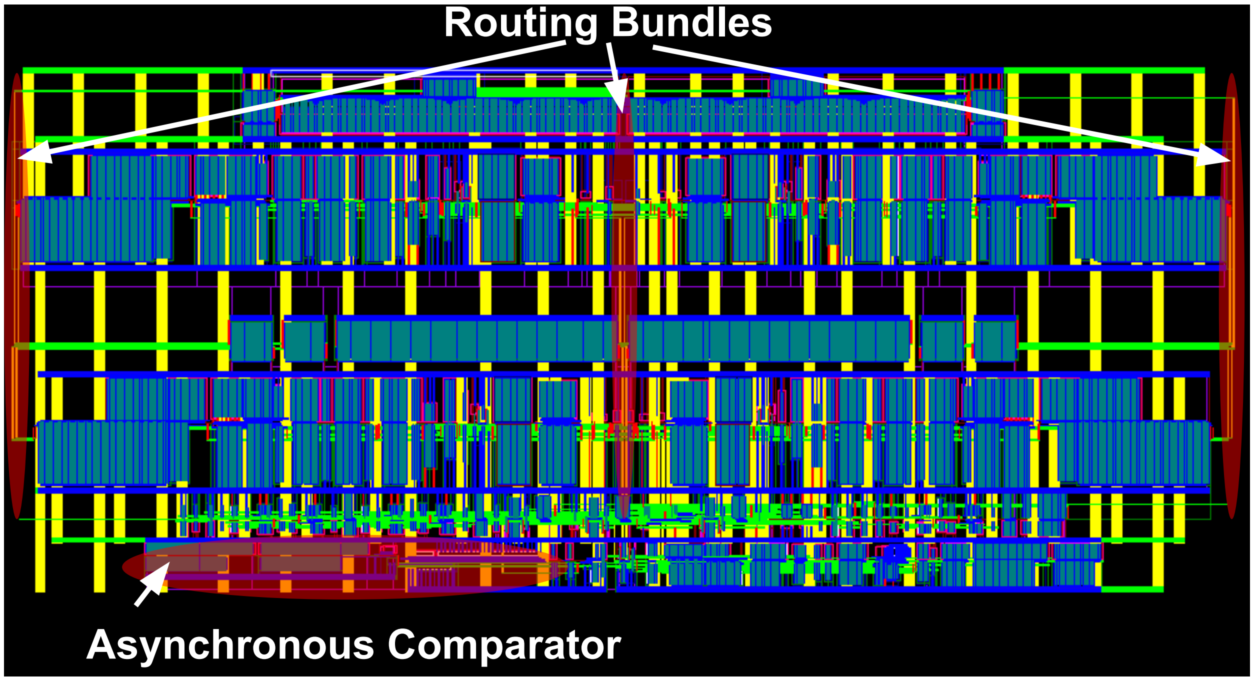


Figure 4.8: Control Block PyCell.

4.2.1.3 Switch Network Generator

The switch network circuit, shown in Figure 4.9, consists of all of the power switches used in the SC converter as well as the first stage of digital logic driving the gates of the power switches. In order to enable decreased gate switching losses under light load conditions, the power switches are partitioned into 4 equal segments, each controlled by a separate driver. The power switches are sized based on the results of a high level optimization script, and the gate capacitors of these switches are used to size the driver gates for an optimal target effort. This target effort is passed from the higher level SC interleaved phase generator that has information about the entire critical path (and not just the final stage).

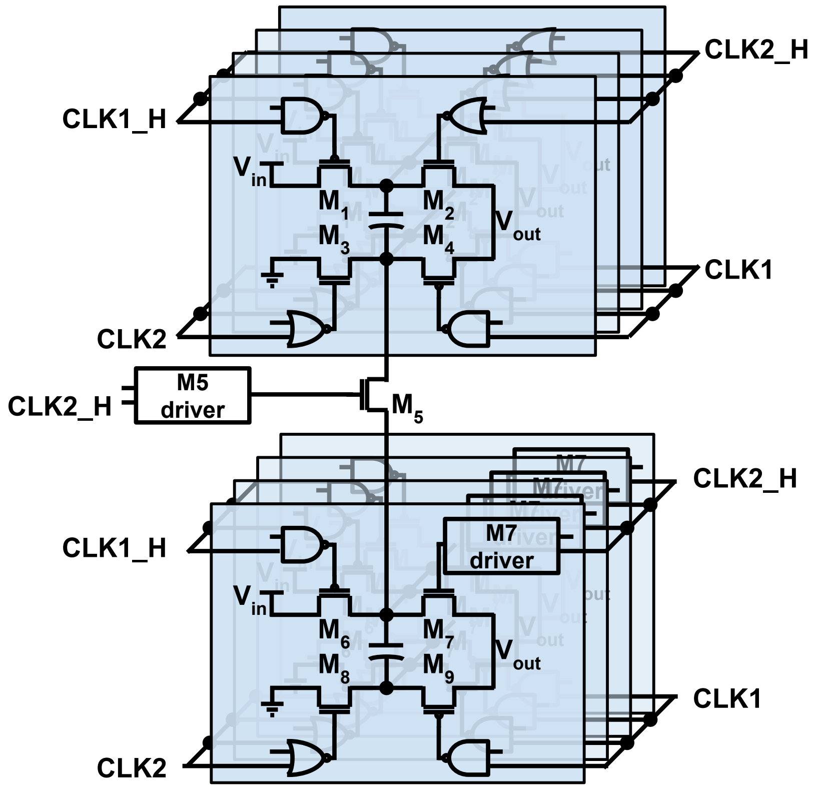


Figure 4.9: Power switches and drivers.

The layout of the switch network could be implemented using the *Standard Block* style, but we chose to implement it as a custom PyCell in order to ensure a compact layout and to avoid potential electromigration issues. Each partitioned switch is implemented as a PyCell like the one shown in Figure 4.10(a) in order to limit the finger width of the switches and maintain a strong connection to the gate. The drivers for most of the switches are NOR and NAND gates created using the *Standard Cell* layout style. In order to enable the switch network to reconfigure from a conversion ratio of 1/2 to 2/3, two of the switches, M5 and M7, require more complex drivers (see the M7 driver schematic in Figure 4.10(b)). As shown in Figure 4.11(d), the M5 driver has twice the vertical pitch of the other cells. Therefore, in order to avoid potential electromigration issues due to excessively large finger widths, its layout is implemented using the *Standard Block* style, as shown in Figure 4.10(c), to connect several vertically stacked copies of a unit element of the M5 driver in parallel.

Another challenge for the switch network layout is the need to deal with multiple voltage domains. It is difficult to handle the isolated pwells and deep nwell boundaries used to create separate voltage domains in a way that can easily scale with different switch widths, driver sizes, and target aspect ratios. The relative locations of the switch/driver groups were chosen to minimize the total number of deep nwell regions required in order to minimize the penalty that results from the large minimum spacing requirement associated with the deep nwell layer.

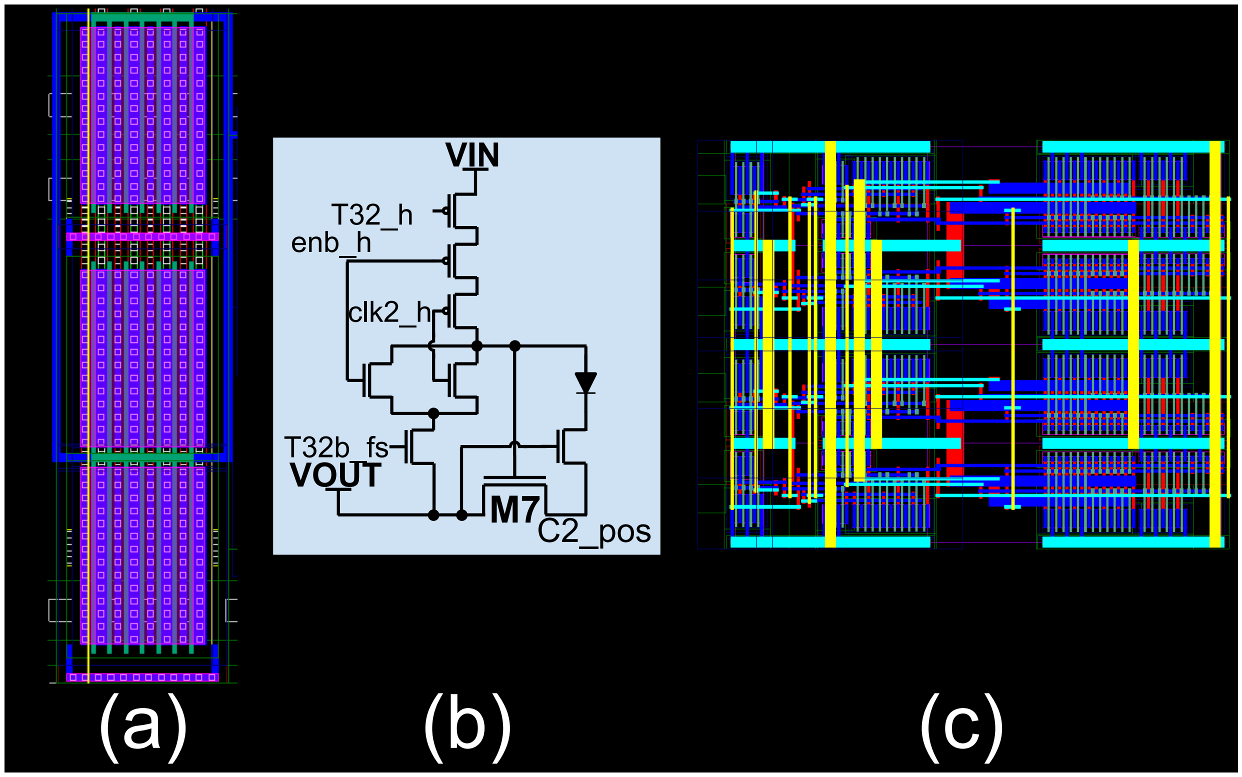


Figure 4.10: Power switch PyCell (a), M7 driver schematic (b), and M5 driver PyCell (c).

The top level layout procedure of the switches and drivers is designed to minimize the amount of empty space in the layout. Using the programmable vertical pitch capability of the driver PyCells and switch PyCell, the bottom row of switches and drivers shown in Figure 4.11 (a) is first placed with a target width set to meet electro-migration specifications. Then a binary search is performed on the pitch of the top row (Figure 4.11(a)-(c)) to match the width of the bottom row. Performing the binary search on the width of the top row while fixing the bottom row is slightly faster than the alternative due to the slower layout generation time of the bottom row resulting from the added complexity of the M7 driver compared to the drivers in the top row. Finally the M5 driver and switch are placed on the left side of the two rows with a vertical pitch set equal to the total height of the two rows (Figure 4.11(d)). This compact layout (see Figure 4.11(e)) would have been hard to obtain using an optimization-based layout generator without specifying a very large number of constraints.

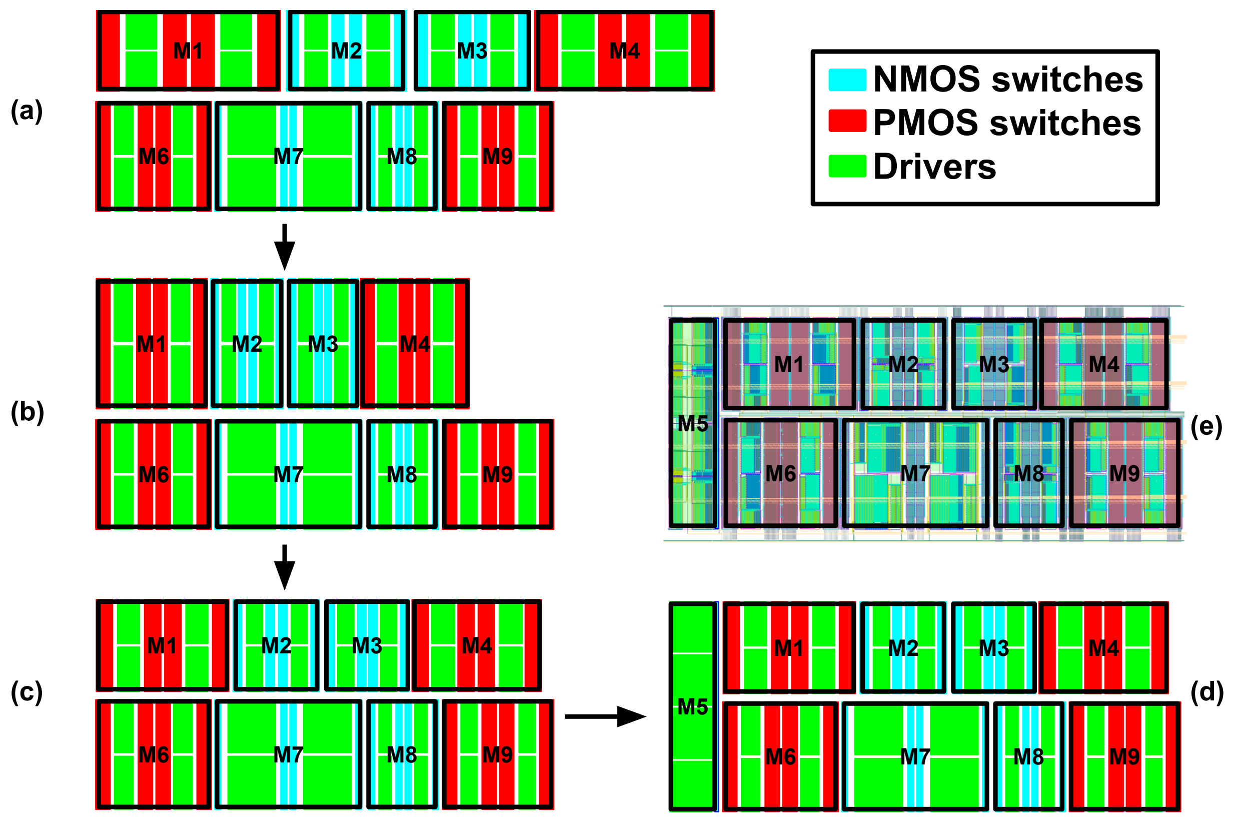


Figure 4.11: A binary search matches the width of the top row of power switches and drivers with the bottom row (a)-(c) before the final M5 switch placed (d). The actual layout is overlaid by the cartoon in (e).

4.2.1.4 Flying Capacitor Generator

The efficiency that a SC converter can achieve for a particular power density is strongly dependent on the density of the flying capacitors. In order to achieve the densest possible capacitor layout, a PyCell was created covering a MOS capacitor with a programmable number of layers of metal finger capacitors. Since the achievable capacitor density is crucial to the optimization of the overall circuit, this PyCell is characterized early on in the generator's design procedures (as described in 2.3.4.1) so that the sizes of the switches and capacitors can be determined before laying out the full converter.

One of the main constraints on the achievable capacitor density is the polysilicon maximum density design rule. In order to achieve the absolute highest density possible, once the full converter PyCell is ready, the density of the MOS capacitor in the flying capacitor PyCell can be optimized using a binary search for the highest density such that the full converter meets the maximum polysilicon density design rule.

4.2.1.5 DAC Generators

The SC converter uses two different DAC generators for calibration and testing purposes. Both generators use the *ArrayModule* class described in 2.3.4.3 to enable the variable structure schematic required to generate a DAC with a variable number of bits. They also both use the *Array* style PyCell class described in 3.3 to simplify the layout procedure.

4.2.1.5.1 Bias Current DAC

The bias current DAC is used to calibrate the asynchronous comparators in the control block. It needs to be able to provide the proper current to reduce the offset of the comparator to a few mV. Once the asynchronous comparator is sized, the nominal required current can be calculated through simulation; the bias DAC is then sized (using the *size_for()* function described in 2.3.4.1) in order to provide that nominal bias current at the worst-case process corner when programmed with a digital value that is one half of its full scale value. The LSB size, and therefore the number of bits required for the bias DAC, is chosen to be small enough to limit the offset due to the step size to be less than a few mV.

The schematics and layouts for the three unit cells used by the *Array* style bias DAC PyCell are shown in Figure 4.12. The core of the unit cell uses the *Standard Cell* style and then vertical and horizontal routing channels are added on top of it and connected using programmable vias that transform the cell into a current source element, a mirror bias cell, or a dummy cell. The routing channels also include ground shields for the bias and output lines and are designed to be abutted with or without mirroring across the y-axis.

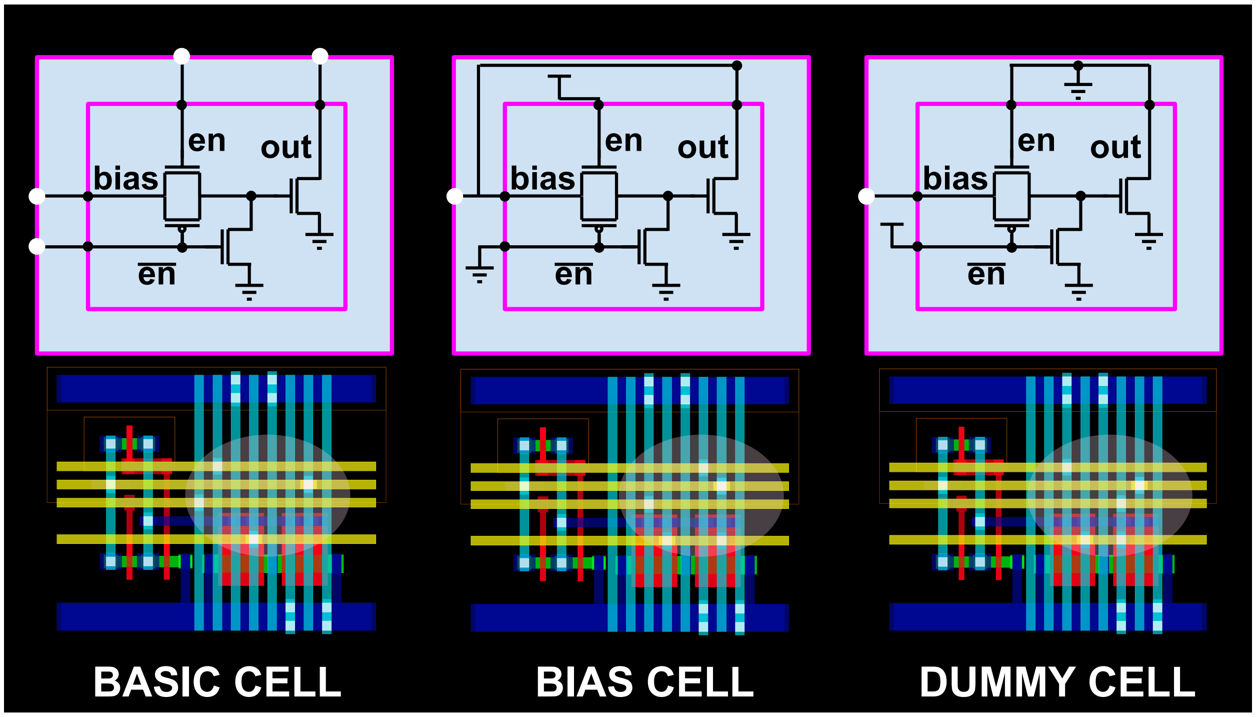


Figure 4.12: Bias DAC unit elements configured as a basic unit cell, a bias cell, and a dummy cell.

There are no high speed signal paths or high current wires in this design so the default configuration of the *Array* style is sufficient for the bias current DAC PyCell. The unit cells from Figure 4.12 are used to build the full array PyCell instances shown in Figure 4.13.

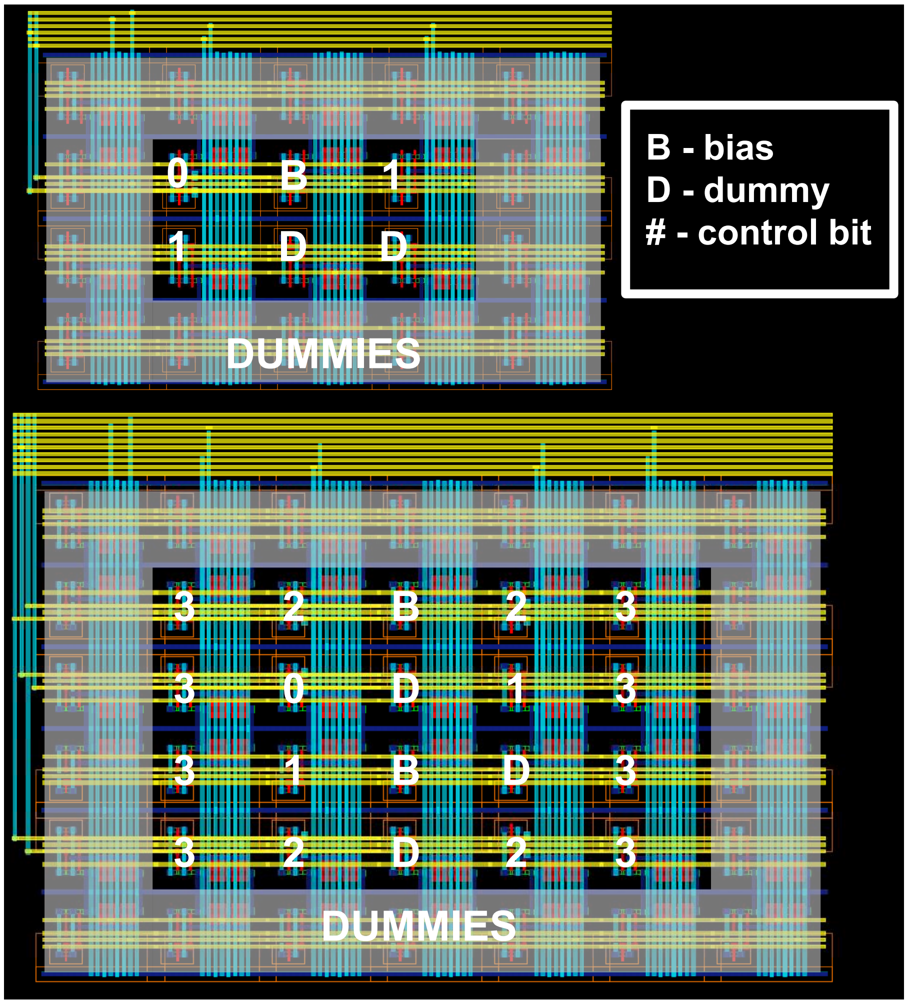


Figure 4.13: Bias Current DAC layout instances for 2 and 4 bit configurations.

4.2.1.5.2 Load DAC

The benefits of fully integrated SC regulators, and in particular the speed of the transient load response, are maximized if the load circuit they are supplying is located on the same die. The parasitic inductance incurred when routing power off the die, through the package, and to a load circuit often precludes the sub-nanosecond response time that is one of the goals of this SC regulator. In order to mimic the fast switching transients of a processor core, a dummy load DAC is integrated with each interleaved phase of the regulator. The schematic of the load itself is quite simple; as illustrated in Figure 4.14, it is just an array of unit-sized NMOS devices controlled by a binary-weighted digital code and connected between the power rails. In order to demonstrate the speed of the SC converter, it is useful to have the capability to adjust the speed of the transient load steps. In order to do this, the gates of the NMOS load devices are driven by a set of current-starved inverters whose bias current is digitally programmed using instances of the bias DAC shown previously. The DAC is sized to supply the maximum load current at the peak output voltage specification (1.2 V) under nominal conditions.

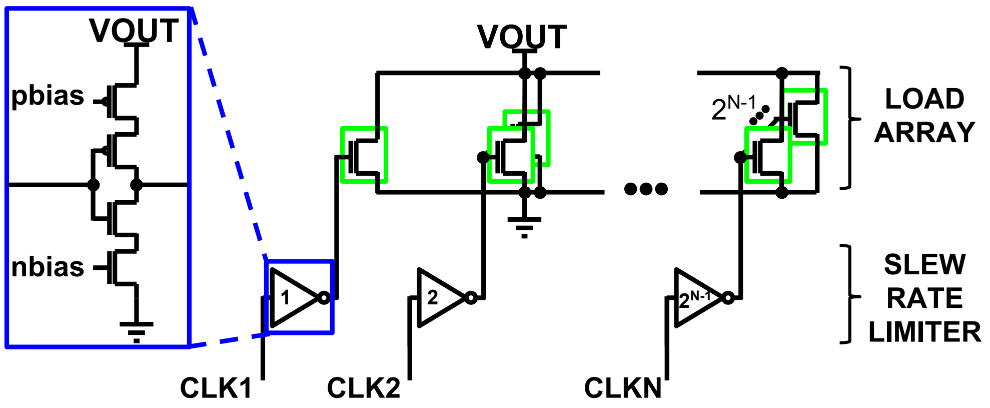


Figure 4.14: Load DAC array and slew rate limiter schematics.

The major design considerations for the load DAC are electromigration and thermal power dissipation. Long channel devices are used to decrease the power density of the load DAC and the DAC itself is distributed with the interleaved phase cells such that each interleaved phase has a single copy of the load DAC. The power rails are also upsized to avoid electromigration problems. Figure 4.15 shows the schematic and layout for the unit cells used in the load array. As described earlier in 3.3, the basic *Array* style pattern dictates that the control signals for even bits are routed to the borders of the array horizontally and for odd bits are routed vertically.

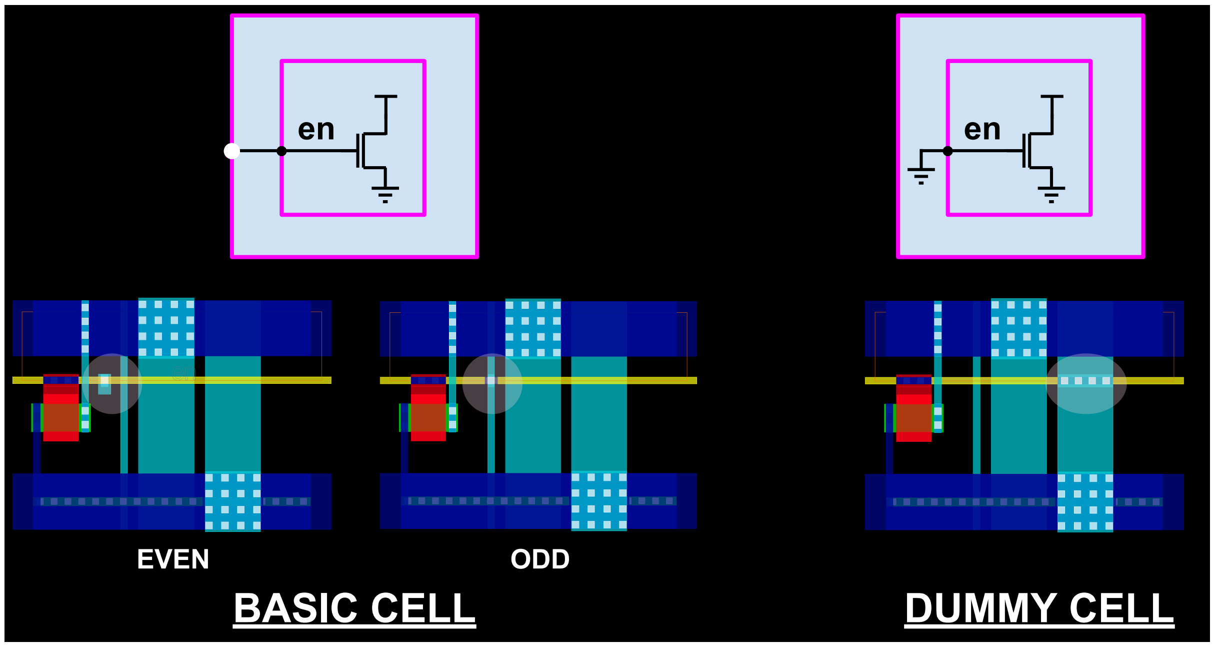


Figure 4.15: Load DAC unit elements configured as a basic unit cell (with an even or odd control bit) and a dummy cell.

Figure 4.16 shows one of the actual load DAC layouts generated for one of the top level converters. A simple wrapper PyCell is created to place the slew rate limiter above the load DAC and connect the routes between them. The power nets of both circuits are connected to the top level power grid later when the load DAC PyCell is instantiated in the SC interleaved phase PyCell.

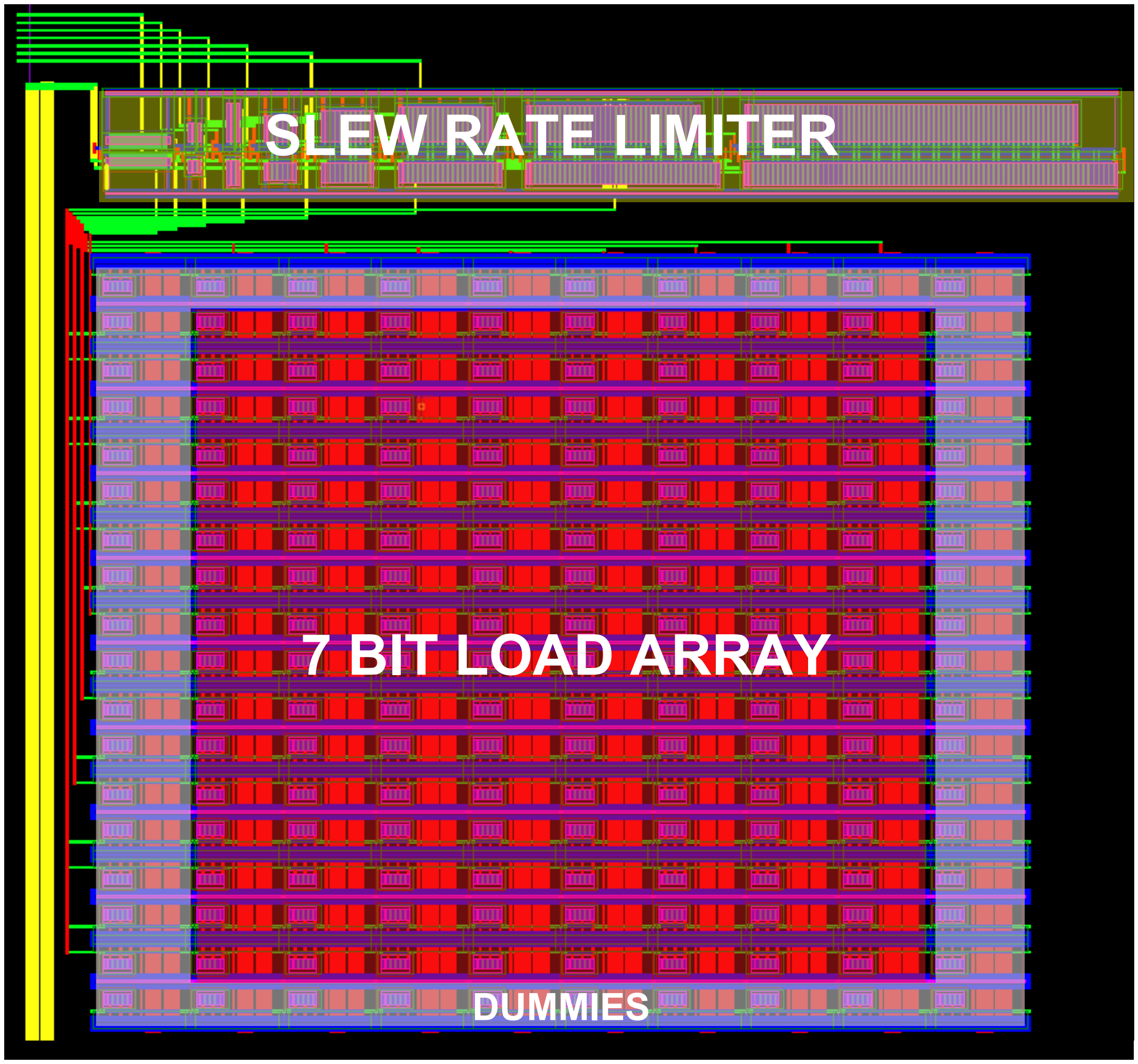


Figure 4.16: Load DAC final layout with slew rate limiter.[10]

4.2.2 Top Level SC Regulator Generator

Based on the analytical optimization flow presented in [41], 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) on switch impedance and capacitance

[10] The connection between the slew rate limiter and the load DAC array shown here is unoptimized. The route shown here was constrainted by the default location of the load array's control pins in the top left corner. As will be dicussed in 5.1, improved routing functions are needed to provide flexible connections that avoid wasting space.

density (see Figure 2.8) as well as target output power specifications. The function then determines the optimal switch and capacitor sizing for the regulator core and the optimal number of interleaved phases. Figure 4.17 shows a block diagram of a six phase SC regulator.

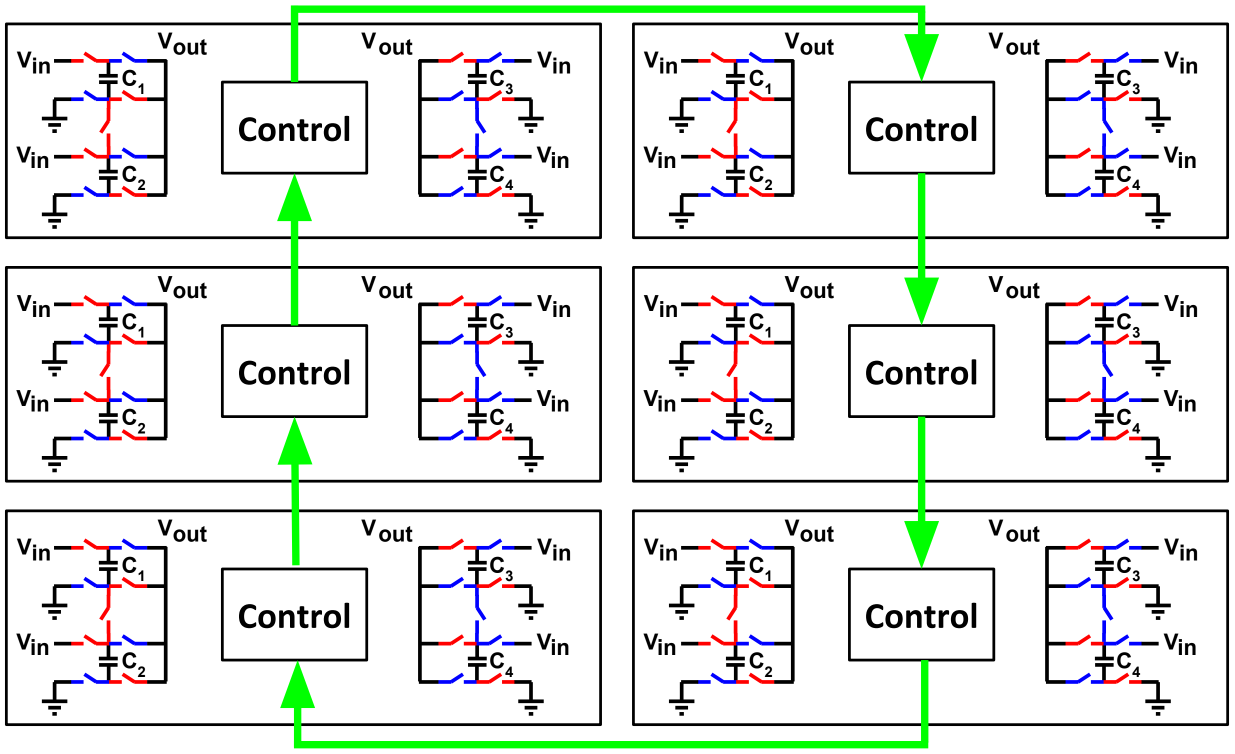


Figure 4.17: 6-phase instance of an interleaved switched-capacitor converter with per-phase lower bound control blocks.

The outputs of the system-level optimization are hierarchically passed to the generators for all of the sub-blocks. Figure 4.18 shows the hierarchical layouts of the three different converters targeting the specifications listed in Table 4.1. The top-level regulator generator inherits from the *ArrayModule* helper class (2.3.4.3) to instantiate the desired number of interleaved phases.

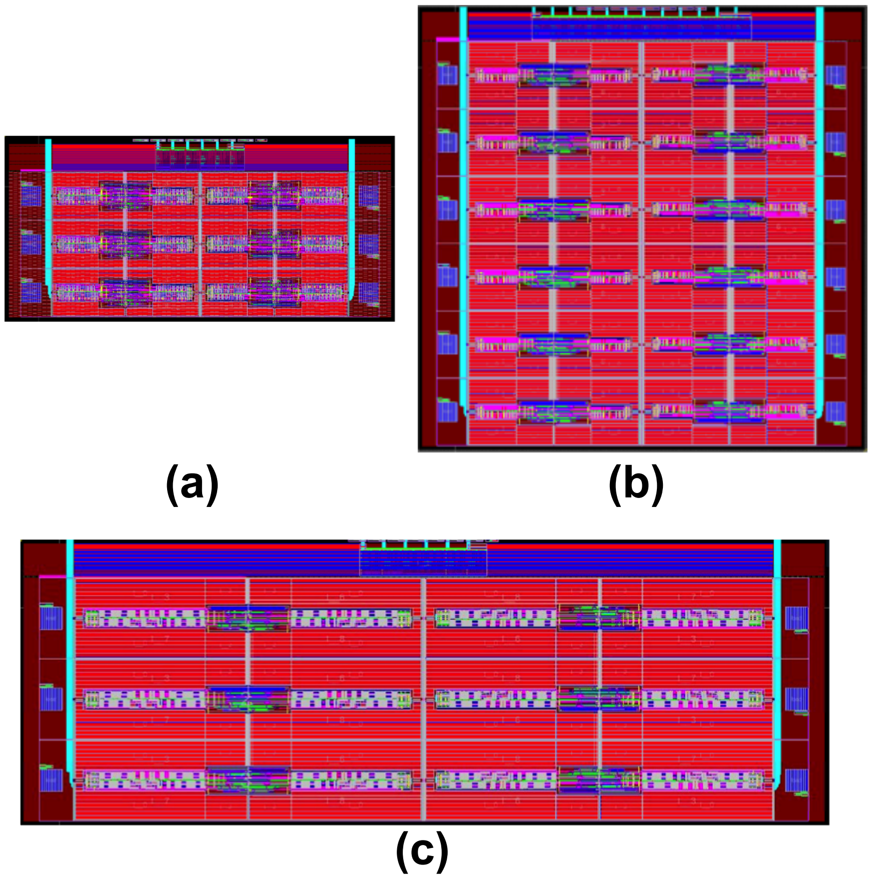


Figure 4.18: Final layouts of three complete SC regulator designs.

After the first iteration of the design flow has been completed, pre (see Figure 4.19 for the top-level schematic simulation of the 1 W regulator responding to load current steps) 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. 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.

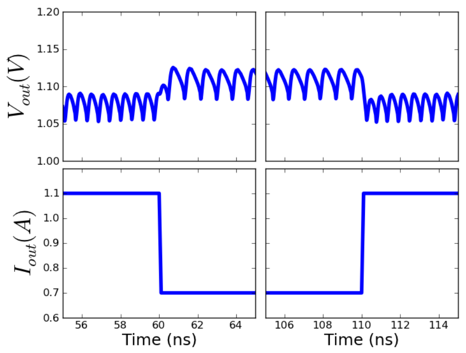


Figure 4.19: Load transient step response simulation result.

4.2.3 Digital Interface

So far, we have focused solely on AMS circuit generators and neglected the fact that most AMS circuits at some point must interface with a digital circuit. In the case of the SC converter, this digital circuit contains a register file of configuration bits that can be written from off chip through a serial interface as well as the digital logic necessary to control the load DAC described in 4.2.1.5. In order to handle the physical interface between these two blocks, the SC converter PyCell creates a file that stores the coordinates and shapes of the top level pins as well as the coordinates of the cell's border. This information is passed as an input to the place and route script used to generate the layout for the synthesized digital block, and the two cells can be instantiated together and aligned such that all the connections are made in a DRC and LVS clean way as shown in Figure 4.20.

The electrical interface between the blocks is handled through careful overdesign of the drive strength of signals on the digital side, and by putting local buffers on most signals in the AMS block right next to the point where they connect to the digital block. This works for the case of this SC converter generator but it is not a general solution. In the future, it would be simple to create a function to automatically extract the post-layout capacitance for all the pins of the AMS blocks and pass that information to the digital tool in the same manner the pin locations are currently passed. Creating a truly general solution for connecting AMS generators and synthesized digital blocks would require substantial additional work, but through careful partitioning of the two domains to minimize the interactions, this work can be substantially reduced.

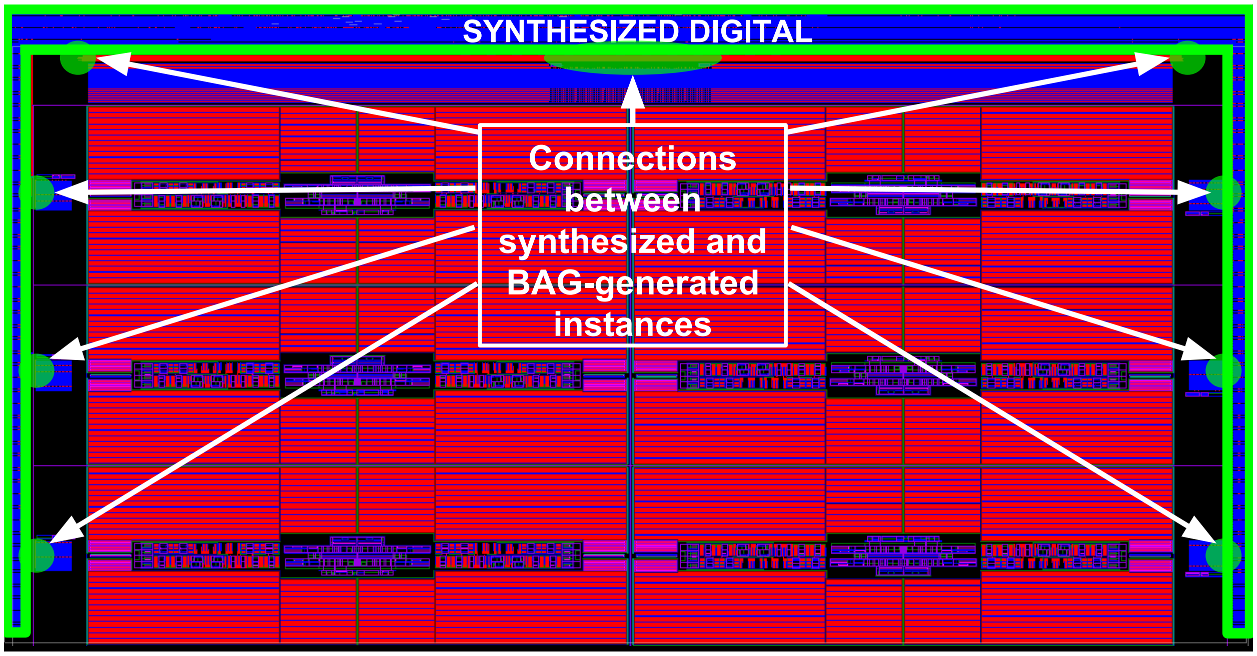


Figure 4.20: Generated AMS instance combined with a synthesized digital block.

Chapter 5 – Conclusion

The number and complexity of analog/mixed-signal circuits included in modern SoC designs is rising, and these circuits are increasingly becoming the primary critical bottleneck gating chip tapeouts even though they often occupy only a small portion of the total die area. In order to address this issue, we argued the need for a more automated methodology in AMS circuit design – specifically, that designers should no longer be focused on producing a sized schematic and layout of a singular circuit instance, but instead should produce circuit generators that can procedurally create schematics and layout of a circuit according to a set of input specifications utilizing the designer's codified knowledge. Therefore, to ease the creation of these AMS circuit generators, we implemented a collection of Python-code called the Berkeley Analog Generator (BAG).

In order to reduce the learning curve for creating generators and to ease the adoption of the BAG framework, we borrowed heavily from the familiar manual design flow – especially for the initial steps of schematic and testbench-entry, which are nearly identical to the traditional flow. We chose Python as a user-friendly implementation language, and tried to structure the framework around an interactive flow that provides designers with an intermediate design exploration step after conceptualizing a new architecture, but before being tasked with writing code for a generator. While developing generators using the BAG framework, we noticed many design procedures relied on pre-characterizing certain key portions of the circuit. In this thesis, we demonstrated helper classes that assist with the creation and maintenance of simulation-based characterization information for device primitives as well as user-created cells. Other helper classes were created to enable circuits with variable structures and to assist with common sizing techniques like the method of logical effort.

We demonstrated the completeness of the BAG framework, i.e. that it can be used for a complete AMS design flow, with an integrated switched-capacitor regulator generator – a complex hierarchical system relevant to modern SoC architectures. By building hierarchical parameterized layouts, we were able to create a top level parameterized layout for the SC regulator and demonstrated its ability to handle different target aspect ratios and sizes. Initially we planned to use commercial AMS place-and-route tools driven by constraints for higher level layout generators because we assumed that it would be frustratingly difficult to create parameterized layouts for cells with many levels of hierarchy. However, as we added more functionality to the layout styles and built more complex manual PyCells for the SC converter, we found that we preferred the level of control that came from using PyCells as compared to a constraint-based automatic tool. It was very useful to be able to reuse the placement and routing methods in the *Standard* styles while still being able to add customized procedures for certain cells on top of the base style.

We also showed that, through judicious use of the PyCell correct-by-construction API and by constraining the layout to fit into one of the BAG layout *styles,* layout generators can be made portable from one process to another with few changes to the underlying code. This portability depends on the new process having a similar number of metal layers and generally similar design rules to the old process. If these requirements are met, even complex hierarchical blocks can be compiled for different processes using single PyCell source file.

5.1 Future Work

The BAG framework is essentially functionally complete in that it can be used to create generators for any kind of AMS circuit. Currently however, a generator writer implementing a design using the BAG framework might need to invest significant effort to e.g. use their preferred design or layout style because it is not yet enabled in the BAG framework. They might also encounter a problem related to a constraint imposed on the design stemming from the use of one of the helper functions or styles that are already in BAG. The potential improvements suggested in this section address one of these two problems by enabling more design techniques with the BAG framework or improving the speed and usability of those already in the framework.

In section 2.3.4.4, we described the BAG framework's handling of veriloga blocks for use in testbench circuits, but it would be useful to have a more complete mixed-mode simulation capability available from within BAG. SPICE level simulation of complex analog circuits can be computationally expensive and have lengthy run times, so one common solution AMS designers employ is to simulate key blocks at the full SPICE level of accuracy while other blocks (e.g. control blocks) are simulated with faster methods using event-driven or behavioral models written in verilog or verilogams respectively. There are many tools for running mixed-mode simulations of this sort, and one can imagine a simple interface in BAG whereby each *Module* could have a default simulation method associated with it that could be adjusted as necessary.

Another simple improvement that could be easily implemented is a function for automatically calculating certain electrical parameters from post-layout parasitic extractions or post-layout simulations. For example, it would be useful to have a property associated with each pin of a *Module* that could return the effective capacitance seen looking into that pin which could be used for interfacing with digital synthesis tools or sizing chains of digital logic. The value could be calculated by recursively finding the capacitance of all connected pins and parsing the extracted netlist for parasitic wiring capacitances, or – depending on the accuracy desired – it could be calculated from simulation of the extracted netlist. All the necessary tools to implement this feature are already in BAG, but have not yet been assembled into a simple-to-use function.

The existing *Standard* styles are quite powerful in terms of enabling designers to create layouts with flexible aspect ratios that can implement most structures that appear in AMS circuits. However, there are several areas for improvement. Routing at the *Standard Cell* level is limited to only the lower two metal layers and the user must be careful to order the horizontal routing channels to avoid any shorts between vertical routes from the bottom of the cell to a higher horizontal routing channel as well as vertical routes from the top of the cell to a lower horizontal routing channel. Currently this limitation is avoided by creating extra horizontal routing channels for a single net, but the ideal solution is a more complex routing mechanism that can handle crossovers. Another limitation in *Standard Cell* is the inability to route from a device pin to a routing channel within a routing group that is not immediately adjacent to the device pin in the vertical stack. This precludes making layouts that look like the one in for the comparator in Figure 5.1 where there are PMOS precharge devices connected to nearly every internal node of the circuit that has five stacked device groups. This layout is very similar to a *Standard Cell* PyCell but is actually a fully custom PyCell because the designer wanted to avoid the current limitations of *Standard Cell*.

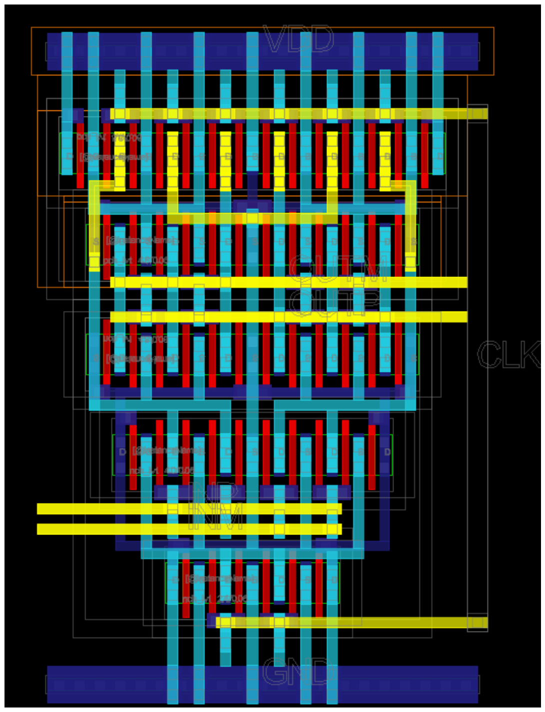


Figure 5.1 Manual PyCell of a comparator.

Another potential improvement for the *Standard* layout styles is to incorporate electromigration and thermal awareness into the PyCells. Currently, the geometric constraints for electromigration are calculated in custom code snippets within specific circuit generators (e.g. the power switches of the SC converter) once the initial sizing is determined. A more general method could be created where electrical information is passed to the layout by default. For example, the maximum current through every device could be stored in the *Module* and passed along with the netlist information to the PyCell. The *Standard* styles could then take this into account at each phase of the layout procedure and save the designer a lot of effort.

A final avenue for expanding the BAG framework is related to the restricted design rules that occur in more advanced technology nodes. In general, the complexity of code required to implement specific layout styles can be greatly simplified by drastically reducing the allowable design space through enforcing very strict routing grids and device placement grids. In older technologies, the performance penalty paid for oversimplifying the layout could be large, but in newer processes with multiple lithographic patterning steps for the lower layers, the DRC rules are already beginning to force extremely regular grid structures. Thus, the penalty for further restrictions might be small enough to be outweighed by the potential gains from the speedup in design iterations. If the grids are regular enough, layout could be performed on abstracted data structures expressed as a set of 2-dimensional arrays, i.e. one for each process layer. By characterizing the regular grid structure, post-layout parasitics could perhaps be estimated very quickly and with high accuracy. Writing placement and routing functions for such a simple representation of the layout would be much easier, and would hopefully lead to more experimentation on the part of designers. The speed of layout generators could potentially be dramatically increased, which could free designers to spend more time optimizing their designs at the system level.

Appendix A

A.1 BAG Generator Code

This section has generator class listing files for example generators.

A.1.1 Load DAC Array

```python
#!/usr/bin/python
# -*- coding: utf-8 -*-
from BAG.DesignDescription.ArrayModule import ArrayModule
from BAG.DesignDescription.GenericCell import GenericCell as GC
from numpy import floor, log2


class BAG_templates__load_dac_array(ArrayModule):

    """
    """

    def inst_pin_formula(self, pin_name):
        '''''
        Maps an instance pin name to a function that returns the net name.

        Args:
            pin_name: (str) Instance pin name for a single instance of the
                            unit cell in the arry.

        Returns:
            A function that takes two parameters (m,n) and returns a net_name.
            The parameters identify the instance being operated on and the
            net_name should be the net connected to the specified instance pin
            <pin_name> on the instance indexed by (m,n).

        Example:

        Raises:

        BAG USER ACTION REQUIRED: Fill in the inst_pin_dict dictionary.
        The keys to this dictionary are the names of the pins on
        the unit cell (which should be the only cell in the template
        schematic for the array module).  The values for the keys
        are functions.  They can be other functions defined in this
        class or lambda functions.  Either way they must have two parameters
        'm' and 'n' which represent the outer and inner (or row and column)
        iteration indexes that will be used in constructing the array.  These
        indexes are integers which can be used to form the net name that the
        instance pin will be attached to.  See some examples below:

        Examples:
            1 - The corresponding pins on all instances are shorted to the same pins.
                The 'a' and 'z' pins on all the unit cell instances are connected
                to nets named 'in' and 'out'pins
            inst_pin_dict = {'vdd': lambda m,n: 'vdd',
                             'a': lambda m,n: 'in',
                             'z': lambda m,n: 'out',
                             'vss': lambda m,n: 'vss'}

            2 - The 'a' and 'z' pins on each unit cell instances are connected
                to a net in a bus named 'in' and 'out' respectively
            inst_pin_dict = {'vdd': lambda m,n: 'vdd',
                             'a': lambda m,n: 'in<' + "%d"%(n) + '>',
                             'z': lambda m,n: 'out<' + "%d"%(n) + '>',
                             'vss': lambda m,n: 'vss'}

            3 - The 'a' and 'z' pins on each unit cell instances are connected
                to a net in a bus named 'in' and 'out' respectively.  The number
                of cells connected to each net in the bus increases by 2 for each
                additional net in the bus (one instance on net 'in<0>', two instances
                on net 'in<1>', 4 instances on net 'in<2>' and so on.
            inst_pin_dict = {'vdd': lambda m,n: 'vdd',
                             'a': lambda m,n: 'in<' + "%d"%(floor(log2(n+1))) + '>',
                             'z': lambda m,n: 'out<' + "%d"%(floor(log2(n+1))) + '>',
                             'vss': lambda m,n: 'vss'}
        '''

        inst_pin_dict = {'control': lambda m, n: 'control',
                         'VSS': lambda m, n: 'VSS', 'VDD': lambda m, n: \
                         'VDD'}

        return inst_pin_dict[pin_name]

## -------------------  Example 1 for defining a tree structure ------------
## This example assumes pin names 'in', 'out', 'vdd', and 'vss'.  Modify as necessary.

## Comment out the lines above defining inst_pin_dict and returning inst_pin_dict[pin_name]
## and uncomment this block of code as well as the other 2 blocks below labeled with 'Example
## 1 for defining a tree structure'
#         inst_pin_dict =  {'vss': lambda m,n: 'vss',
#                           'vdd': lambda m,n: 'vdd',
#                           'a': self.instance_input_net,
#                           'z': self.instance_output_net}
#
#         return inst_pin_dict[pin_name]
#
#     def instance_output_net(self,m,n) :
#         Serves same purpose as
#         if m == self.dim_0.value-1 : # last iteration
#             return 'out'
#         else :
#             return 'int_' + "%d"%(int(m+1)) + '_' + "%d"%(int(n/self.radix.value))
#
#
#     def instance_input_net(self,m,n) :
#         if m == 0 : # first iteration
#             return 'in<' + "%d"%(int(n)) + '>'
#         else :
#             return 'int_' + "%d"%(int(m)) + '_' +  "%d"%(int(n))
      ## -------------------  Example 1 for defining a tree structure ------------

          def pin_formula(self, pin_name):
              '''''
              Maps the cell pin names to functions that returns the net name.

              Args:
                  pin_name: (str) A cell pin name at the top level of the array cell.

              Returns:
                  A function that takes no parameters and returns a net_name.

              Example:

              Raises:

              BAG USER ACTION REQUIRED: Fill in the pin_dict dictionary.
              The keys to this dictionary are the names of the pins on
              the array cell (which should be the pin symbols in the template
              schematic for the array module).  The values for the keys
              are functions.  They can be other functions defined in this
              class or lambda functions.  These functions can access the parameters
              of the array class including 'dim_0' and 'dim_1' which are the default array
              iteration indexes.  The values returned by these functions
              are the net names that should be connected to the pins.  The pins will
              be renamed so their names match the net names.

              Examples:
                  1 - The corresponding pins on all instances are shorted to the same pins.
                      The 'a' and 'z' pins on all the unit cell instances are connected
                      to nets named 'in' and 'out'pins
                  pin_dict =  {'vdd': lambda : 'vdd',
                               'in':  lambda : 'in',
                               'out': lambda : 'out',
                               'vss': lambda : 'vss'}

                  2 - The 'a' and 'z' pins on each unit cell instances are connected
                      to a net in a bus named 'in' and 'out' respectively
                  pin_dict =  {'vdd': lambda : 'vdd',
                               'in':  lambda : 'in<' + "%d"%(self.dim_1.value-1) + ':0>',
                               'out': lambda : 'out<' + "%d"%(self.dim_1.value-1)  + ':0>',
                               'vss': lambda : 'vss'}
              '''

      ## -------------------  Example 1 for defining a tree structure ------------
      ## This example assumes pin names 'in', 'out', 'vdd', and 'vss'.  Modify as necessary.
      ## Comment out the  lines below defining pin_dict and uncomment this code block 
      ## as well as the other 2 blocks above/below labeled with 'Example 1 for defining a tree structure'
      #         pin_dict =  {'vss': lambda : 'vss',
      #             'in':  lambda : 'in<' + "%d"%(int(2**self.dim_0.value-1)) + ':0>',
      #             'out': lambda : 'out',
      #             'vdd': lambda : 'vdd',
      #             'vss': lambda : 'vss'}
      ## -------------------  Example 1 for defining a tree structure ------------

              pin_dict = {'control': lambda : 'control', 'VSS': lambda : \
                          'VSS', 'VDD': lambda : 'VDD'}
              return pin_dict[pin_name]

          def testbench_formula(self):
              '''
              How to alter the testbench instances and their inst_pins so they
              can connect properly to the new
              testbench_pin_dict = { 'DNL' : { 'vdd': lambda 'vdd'
                                               '   }
              '''

              pass

              # testbench_pin_dict =

          def instance_formula(self, m, n):
              '''''
              BAG USER ACTION SUGGESTED: Formula generates the suffix
              for the iterated unit cells.
              '''

              return '%04d' % int(m) + '_' + '%04d' % int(n)

          def __init__(
              self,
              prj,
              parent=None,
              in_testbench=False,
              ):
              generic_cell = GC('load_dac_array', 'BAG_templates')
              ArrayModule.__init__(self, prj, generic_cell, parent,
                                   in_testbench)

              # self.dim_0 must be an integer but self.dim_1 can be either an integer or a function that
              # returns an integer.

              self.dim_0 = 1
              self.dim_1 = 1


              # #  -------------------  Example 1 for defining a tree structure ------------
              # # This example assumes pin names 'in', 'out', 'vdd', and 'vss'.  Modify as necessary.
              # # Comment out the 2 lines above defining self.dim_0 and self.dim_1 and uncomment this code block
              # # as well as the other 2 blocks above labeled with 'Example 1 for defining a tree structure'
              # self.radix = self.add_design_parameter(DP(yaml_param_dict='radix',module_handle=self,callback=self.update_array))
              # self.radix = 4
              # self.num_bits = self.add_design_parameter(DP(yaml_param_dict='num_bits',module_handle=self,callback=self.update_array))
              # self.num_bits = 4
              # self.dim_0.set_expression('int(numpy.ceil(self.module.num_bits.value/numpy.log2(self.module.radix.value)))')
              # self.dim_1 = lambda outer_iter: int((2**self.num_bits.value)/(self.radix.value**outer_iter))
              # #  -------------------  Example 1 for defining a tree structure ------------

              # # -------------------  Example 2 for defining a tree structure ------------
              # self.dim_1 = lambda outer_iter: (2^outer_iter.value) #Use this definition for self.dim_1 if the second dimension
```

```
                #                                                    #of the array depends on the level as in a decoder or other
                #                                                    #tree-like circuit.  The function you write should take a single
                #                                                    #parameter representing the outer iteration variable
                #            outer_iter = 0             #           inner_iter = 2^0 = 1
                #                                     /  \
                #            outer_iter = 1          #    #         inner_iter = 2^1 = 2
                #                                  / \   / \
                #            outer_iter = 2       #   # #   #       inner_iter = 2^2 = 4
                # # -------------------  Example 2 for defining a tree structure ------------
```

A.1.2 DCDC Phase Unit Cell Generator

```
#!/usr/bin/python
# -*- coding: utf-8 -*-

##------------------------------------------------------------
## import modules
##------------------------------------------------------------

"""
MODULE BAG_templates__conv_unit_cell2_wocap

Fill in the funciton of this design module here


"""

from BAG.DesignDescription.VarStructDesignModule import VarStructDesignModule
from BAG.DesignDescription.GenericCell import GenericCell as GC
from pylab import ceil
from copy import copy, deepcopy
import pdb
from BAG.DesignDescription.logical_effort import logical_effort
import yaml


##------------------------------------------------------------
## class DesignModule
##------------------------------------------------------------

class BAG_templates__conv_unit_cell2_wocap(VarStructDesignModule,
    logical_effort):

    """
    """

    def __init__(
        self,
        prj,
        parent=None,
        in_testbench=False,
        ):
        '''''
        If project_name is empty then a new user_project will be created
....(in both the CAD and python directories)
        '''

        generic_cell = GC('conv_unit_cell2_wocap', 'BAG_templates')
        VarStructDesignModule.__init__(self, prj, generic_cell, parent,
                in_testbench)

        # self.force_schematic = True

        logical_effort.__init__(self)
        self.width_finger_pairs = [['M5_finger_w', 'M5_fingers']]
        self.rBulk = 1.0
        self.areaUnitCap = 1.0
        self.interleaving = 1
        self.HoverW = 0.0
        self.route4LVS = 'False'
        self.widthM5 = 0.0
        self.cachedParameterFile = ''
        self.path2CompCore = ''
        self.powerLevel = ''
        self.default_pycell_lib_cell = \
            'demo_cells_oa_lib/DCDC2OutOfPhaseWoCaps'

    def simple_size(self):
        pass

    def set_default_sizes(
        self,
        g_mult=1,
        g_width=None,
        g_length=None,
        g_pnrat=None,
        vin=2.0,
        vref=0.9,
        unit_inv_cap=1e-15,
        switch_load_cap=2.0e-12,
        target_fanout=4.0,
        nonoverlap_ratio=2.0,
        ):

        ccell = self
        prj = self.prj

        # size the switches

        gate_cap_ff_per_um = 1.0  # extract this number
        total_switch_width = switch_load_cap / 1e-15 \
            / gate_cap_ff_per_um * 1e-6
        for the_cell in [ccell.I19.I0, ccell.I20.I0]:
            for the_cap in [the_cell.C1, the_cell.C2]:
                the_cap.set_parameters('W', 500e-6)
                the_cap.cap_mult = 10
                the_cap.mult = 280
            the_cell.p_switch_width = g_width * 10
            the_cell.n_switch_width = g_width * 10
            the_cell.n_switch_mult = int(total_switch_width
                    / the_cell.n_switch_width.value)
                   the_cell.p_switch_mult = int(total_switch_width
                           / the_cell.p_switch_width.value)
                   the_cell.cfly = 2.8e-10
```

```
                    the_cell.cfly_ic = vref

                unit_inverter_gate_cap = g_width * 3 * gate_cap_ff_per_um / 1e-6
                switch_gate_cap = ccell.I20.I0.n_switch_width.value \
                    * the_cell.n_switch_mult.value * gate_cap_ff_per_um / 1e-6
                switch_load = int(switch_gate_cap / unit_inverter_gate_cap)
                if not g_width:
                    dummy_M = prj.classes['BAG_prim__nmos4'](prj)
                    dummy_M.W = 0.0
                    g_width = 2.0 * dummy_M.W.value
                    del dummy_M
                if not g_length:
                    dummy_M = prj.classes['BAG_prim__nmos4'](prj)
                    dummy_M.L = 0.0
                    g_length = dummy_M.L.value
                    del dummy_M
                if not g_pnrat:
                    g_pnrat = 1.6
                ccell.set_parameters('PN_ratio', g_pnrat)
                ccell.set_parameters('L', g_length)
                ccell.set_parameters('W', g_width)
                ccell.set_parameters('stack', 1)

                # set capacitor cell driver sizes

                ccell.I20.pnrat = g_pnrat * 2  # multiply by 2 because this is a nor gate
                ccell.I20.pgatedrive_W = g_width
                ccell.I19.pnrat = g_pnrat * 2  # multiply by 2 because this is a nor gate
                ccell.I19.pgatedrive_W = g_width
                gatedrive_mult = ceil(switch_load / (target_fanout * 3 / 5.0))  # logical effor of nor is 3/5
                ccell.I20.pgatedrive_mult = gatedrive_mult
                ccell.I19.pgatedrive_mult = gatedrive_mult

                ccell.sw_cap = switch_load_cap
                ccell.critical_path.set_default_sizes(g_mult, g_width,
                        g_length, g_pnrat)
                ccell.critical_path.adjust_mults(gatedrive_mult,
                        target_fanout=target_fanout,
                        nonoverlap_ratio=nonoverlap_ratio)

                # these phase_splitters drive clock of the toggle flop

                load_mult = ceil(ccell.critical_path.dflop.pg_mult.value
                                 + ccell.critical_path.dflop.tristate_mult_2.value
                                 + ccell.critical_path.dflop.tristate_mult.value)
                ccell.phase_split_long.size_to_drive(load_mult, target_fanout)

                # comparator should actually sized based on delay and mistmatch constraints

                mult_normalization_factor = \
                    ccell.phase_split_long.inverting_stage.W.value \
                    / ccell.phase_split_long.buf_stage_1.W.value
                ps_mult = ccell.phase_split_long.buf_stage_1.mult.value \
                    + mult_normalization_factor \
                    * ccell.phase_split_long.inverting_stage.mult.value
                comp_mult = ceil(ps_mult / target_fanout)
                ccell.comparator.set_default_sizes(comp_mult, g_width)
```

```python

        # size gates in clk_output driver
        # should be sized based on comparator

        clk_driver_mult = 4
        ccell.pull_up_keeper.mult = clk_driver_mult * 2
        ccell.pull_down_keeper.mult = clk_driver_mult
        ccell.pull_down_reset.mult = clk_driver_mult
        ccell.pull_up_long_1.set_default_sizes(g_mult=g_mult,
                g_width=g_width, g_length=g_length, g_pnrat=g_pnrat)
        ccell.pull_up_long_2.set_default_sizes(g_mult=g_mult,
                g_width=g_width, g_length=g_length, g_pnrat=g_pnrat)
        ccell.pull_up_long_1.pull_up_mult = clk_driver_mult * 2
        ccell.pull_up_long_2.pull_up_mult = clk_driver_mult * 2

        # make sure reset parameter uses thick gate devices so it can handle full
        # scale voltages on its inputs

        ccell.reset_inv_fs.set_parameters('device_intent',
                'high_voltage')
        dummy_M = prj.classes['BAG_prim__nmos4'](prj)
        dummy_M.device_intent = 'high_voltage'
        dummy_M.W = 0.0
        dummy_M.L = 0.0
        the_width = dummy_M.W.value
        the_length = dummy_M.L.value
        del dummy_M
        ccell.reset_inv_fs.W = the_width
        ccell.reset_inv_fs.L = the_length

        # pnration should be fine

    def get_pycell_to_bag_mapping(self, lib_cell):
        if lib_cell == 'demo_cells_oa_lib/DCDC2OutOfPhaseWoCaps':
            pycell_to_bag_mapping = [['control', self.control_core,
                    'BAG_std_cells_oa/comp_core2'], ['load',
                    self.load_dac, 'BAG_std_cells_oa/load_dac'], ['slew'
                    , self.load_dac.slew_rate_limiter,
                    'BAG_std_cells_oa/slew_rate_limiter'], ['switches',
                    self.swcap_w_driver_out_phase,
                    'demo_cells_oa_lib/DCDC_unit_cell_noM5']]

                                      # ['caps',, 'demo_cells_oa_lib/dense_unit_capNM']

        return pycell_to_bag_mapping

    def pycell_mapper(self, lib_cell, use_pycell_defaults=False):
        '''''
        This function should map the BAG parameters to pycell parameters and return a dictionary
        which can then be passed (through a yaml file) to the pycell
        '''

        the_params = {}
        the_cell = GC(self.cell_name, self.inst_lib_name)

        # the pycell_to_bag_mapping should help map the subpycells to BAG modules
        # each row should be of the format [ pycell_instance_name, bag_module, lib_cell]

        if lib_cell == 'demo_cells_oa_lib/DCDC2OutOfPhaseWoCaps':
            self._dict_for_param_file = {}
            self._dict_for_param_file['instance_parameters'] = dict()
            for (pycell_instance_name, bag_module, lib_cell) in \
                self.get_pycell_to_bag_mapping(lib_cell):

                self._dict_for_param_file['instance_parameters'
                        ][pycell_instance_name] = \
                    bag_module.pycell_mapper(lib_cell)
            self._dict_for_param_file.update(dict(
                rBulk=self.rBulk,
                areaUnitCap=self.areaUnitCap,
                interleaving=self.interleaving,
                HoverW=self.HoverW,
                route4LVS=self.route4LVS,
                widthM5=self.widthM5,
                cachedParameterFile=self.cachedParameterFile,
                loadBits=self.load_dac.bits.value,
                vRefWidth=self.parent_module.parent_module.parent_module.vRefWidth,
                ))
            self.param_file = self.prj.project_directory + '/OOP_' \
                + self.powerLevel + '.yaml'
            yaml_string = yaml.dump(self._dict_for_param_file)
            print 'DCDC2OutOfPhaseWoCaps yaml parameter file %s' \
                % self.param_file
            the_file = open(self.param_file, 'w')
            the_file.write(yaml_string)
            the_file.close()
            the_params = {'param_file': self.param_file}

                          # 'polyDensity': float(self.polyDensity.value),

            the_params.update(self._dict_for_param_file)
        if not use_pycell_defaults:
            return the_params
        else:
            return {}

    def pycell_reverse_mapper(self, result_props):
        '''
        This level doesn't get the control core and switch cell parameters in the result_props

        '''

        self.M5_finger_w = result_props['M5_finger_w']
        self.M5_fingers = result_props['M5_fingers']

        # ['switches', self.swcap_w_driver_out_phase, "demo_cells_oa_lib/DCDC_unit_cell_noM5"],
        # pycell_reverse_mapper for swcap_w_driver_out_phase gets called automaticallys

        # swcap.I0.M1M4_finger_w = result_props['M1M4_finger_w']*1e-6
        # swcap.I0.M1M4_fingers = result_props['M1M4_fingers']
        # swcap.I0.M2M3_finger_w = result_props['M2M3_finger_w']*1e-6
        # swcap.I0.M2M3_fingers = result_props['M2M3_fingers']
```

```
            # swcap.I0.M6M9_finger_w = result_props['M6M9_finger_w']*1e-6
            # swcap.I0.M6M9_fingers = result_props['M6M9_fingers']
            # swcap.I0.M7M8_finger_w = result_props['M7M8_finger_w']*1e-6
            # swcap.I0.M7M8_fingers = result_props['M7M8_fingers']

            # swcap.I0.M1M4_true_mult = result_props['M1M4_true_mult']
            # swcap.I0.M2M3_true_mult = result_props['M2M3_true_mult']
            # swcap.I0.M6M9_true_mult = result_props['M6M9_true_mult']
            # swcap.I0.M7M8_true_mult = result_props['M7M8_true_mult']

        def create_layout_for_lvs(
            self,
            flatten=False,
            use_pycell_defaults=False,
            route4LVS=False,
            ):

            the_cell = GC(self.cell_name, self.inst_lib_name)
            pcell = GC('DCDC2OutOfPhaseWoCaps', 'demo_cells_oa_lib')
            lib_cell = pcell.lib_name + '/' + pcell.cell_name
            if route4LVS:
                self.route4LVS = 'True'
            else:
                self.route4LVS = 'False'
            param_dict = self.pycell_mapper(lib_cell, use_pycell_defaults)

                          # layout : schematic

            pin_dict = {'pmos_bias': 'pmos_slew_bias',
                        'nmos_bias': 'nmos_slew_bias'}

                             # incomplete
            # pin_dict = zip(self.pins.keys(),self.pins.keys())
            # pdb.set_trace()

            result_props = \
                self.prj.db.instantiate_layout_for_extraction(the_cell,
                    pcell, param_dict, pin_dict, flatten=flatten)
            self.result_props = result_props

        def _perform_structure_update(self):
            '''
            The user should implement this function so the structure of the
            module is updated based on the parameter
            values.  It will be called triggered when any structural design
            parameters are altered although it will not actually be run until
            update_structure is called.
            '''

            VarStructDesignModule._perform_structure_update(self)
            self.pins['load_code'].pin_name = 'load_code<%d:0>' \
                % (self.load_dac.bits.value - 1)
            self.pins['load_code'].net_name = 'load_code<%d:0>' \
                % (self.load_dac.bits.value - 1)
            self._update_connections_from_children()
            for x in self.connections:
                if x.net_name == 'load_code':
                    x.net_name = 'load_code<%d:0>' \
                        % (self.load_dac.bits.value - 1)
                if x.pin_name == 'load_code' and not x.inst_name:
                    x.pin_name = 'load_code<%d:0>' \
                        % (self.load_dac.bits.value - 1)

            # self.pin_order = [self.pins[x].pin_name for x in  self._orig_pin_order]

        def _update_connections_from_children(self):
            '''''
            Refresh the connections on this module by starting with the original schematic connections
            (from the yaml file) and then update them according to the current pins object
            in all child instances
            '''

            self.connections = deepcopy(self._original_connections)
            for (inst_name, inst) in self.instances.iteritems():
                for (orig_pin_name, pin) in inst.pins.iteritems():
                    if not pin.pin_name == orig_pin_name:

                        # print inst_name,pin['pin_name'],orig_pin_name
                        # indicates this pin name has been changed due to a structural change
                        # so we should update that in the current parent module

                        for x in self.connections:
                            if x.inst_name == inst_name and x.pin_name \
                                == orig_pin_name:
                                x.pin_name = pin.pin_name
                                x.orig_pin_name = orig_pin_name
                                break  # should only be one pin with that name

        def generate_layout(self):
            '''''Generates the layout of the current Module.  Abstract function
            that should be implemented by all functions that inherit from
            DesignModule.
    ....'''

            pass

        def design(
            self,
            DCDC_params={},
            the_switch_params={},
            num_phases=10,
            L=60e-9,
            target_effort=None,
            cell_height=None,
            switch_device_intent='standard',
            later_stage_device_intent='fast',
            power_scale_factor=0.6,
            **kwargs
            ):
            '''''Abstract function that should be implemented by all functions
        ....that inherit from DesignModule.
        ....'''

            oop = self

            # power_scale_factor = 0.25

```

```
                self.update_device_char()  # There aren't any nmos or pmos at this level so it probably uses defaults
                for arg in (
                    'pnrat',
                    'stack_2n',
                    'cgate',
                    'cgaten',
                    'cgatep',
                    'stack_3n',
                    'stack_2p',
                    'stack_3p',
                    'gamma',
                    'tinv',
                    ):
                    exec "%s = kwargs.get('%s',self.%s)" % (arg, arg, arg)  # put argument into local variables for this function
                oop.set_parameters('pcell_id', 'basic')
                if self.prj.__dict__.has_key('Wmin'):
                    Wmin = self.prj.Wmin
                else:
                    Wmin = 400e-9
                snap = self.prj.snap

                oop.control_core.simple_size()

                if cell_height:
                    oop.control_core.cell_height = cell_height

                # set M5 size

                self.M5_fingers = 1
                self.M5_finger_w = DCDC_params['instance_parameters']['OOP'
                        ]['M5width'] / 1e6
                self.rBulk = DCDC_params['instance_parameters']['OOP']['rBulk']
                self.areaUnitCap = DCDC_params['instance_parameters']['OOP'
                        ]['areaUnitCap']
                self.interleaving = DCDC_params['instance_parameters']['OOP'
                        ]['interleaving']
                self.HoverW = DCDC_params['instance_parameters']['OOP']['HoverW'
                        ]
                self.widthM5 = DCDC_params['instance_parameters']['OOP'
                        ]['M5width']
                self.cachedParameterFile = './OOPParams_' \
                    + DCDC_params['powerLevel'] + '.yaml'
                self.path2CompCore = 'control_core_' + DCDC_params['powerLevel']
                self.powerLevel = DCDC_params['powerLevel']
                self.fix_widths()
                M5_load_cap = self.M5_in_phase.total_width * cgate

                oop.swcap_w_drivers_in_phase.simple_size(the_switch_params=the_switch_params,
                        switch_device_intent=switch_device_intent,
                        M5_load_cap=M5_load_cap)
                oop.swcap_w_driver_out_phase.simple_size(the_switch_params=the_switch_params,
                        switch_device_intent=switch_device_intent,
                        M5_load_cap=M5_load_cap)

                # oop.Cfly_in_phase.simple_size()
                # oop.cfly_out_phase.simple_size()

                cur_volt_pair = [(1.0, 0.8), (1.0, 1.2)]
                oop.load_dac.design(bits=7, cur_volt_pairs=cur_volt_pair,
                                    num_phases=num_phases)

                # figure out which path has the highest fanout

                partitions = the_switch_params['swPartitions']
                nov_ratio = 1.0
                nov_pg_ratio = 1 / 4.0  # pg inputs to are 4 times smaller than what they are driving

                nov_driver = oop.control_core.critical_path.nov_ls.nov_high

                # intrinsic delay factors

                p_nor2 = (pnrat * stack_2p + 2) / (1 + pnrat)
                p_nand2 = (pnrat * 2 + stack_2n) / (1 + pnrat)
                p_flying_inv = (stack_3p * pnrat + 1) / (1 + pnrat)
                p_nov_h = (2 * pnrat + 1) / (1 + pnrat)
                p_nov_l = (pnrat + 2) / (1 + pnrat)
                p_pg = 1

                delay_fanout = 4.0

                # side loads from layout parasitics

                cin_8_side_load = 5e-15

                C1 = power_scale_factor * partitions \
                    * (oop.swcap_w_drivers_in_phase.I0.I0.M1.W.value
                       * oop.swcap_w_drivers_in_phase.I0.I0.M1.mult.value
                       * cgatep)
                C2 = power_scale_factor * partitions \
                    * (oop.swcap_w_drivers_in_phase.I0.I0.M2.W.value
                       * oop.swcap_w_drivers_in_phase.I0.I0.M2.mult.value
                       * cgaten)
                C3 = power_scale_factor * partitions \
                    * (oop.swcap_w_drivers_in_phase.I0.I0.M3.W.value
                       * oop.swcap_w_drivers_in_phase.I0.I0.M3.mult.value
                       * cgaten)
                C4 = power_scale_factor * partitions \
                    * (oop.swcap_w_drivers_in_phase.I0.I0.M4.W.value
                       * oop.swcap_w_drivers_in_phase.I0.I0.M4.mult.value
                       * cgatep)
                C5 = power_scale_factor * oop.M5_out_phase.W.value \
                    * oop.M5_out_phase.mult.value * cgaten
                C6 = power_scale_factor * partitions \
                    * (oop.swcap_w_drivers_in_phase.I0.I0.M6.W.value
                       * oop.swcap_w_drivers_in_phase.I0.I0.M6.mult.value
                       * cgatep)
                C7 = power_scale_factor * partitions \
                    * (oop.swcap_w_drivers_in_phase.I0.I0.M7.W.value
                       * oop.swcap_w_drivers_in_phase.I0.I0.M7.mult.value
                       * cgaten)
                C8 = power_scale_factor * partitions \
                    * (oop.swcap_w_drivers_in_phase.I0.I0.M8.W.value
                       * oop.swcap_w_drivers_in_phase.I0.I0.M8.mult.value
                       * cgaten)
                C9 = power_scale_factor * partitions \
                    * (oop.swcap_w_drivers_in_phase.I0.I0.M9.W.value
                       * oop.swcap_w_drivers_in_phase.I0.I0.M9.mult.value
```

```
                        * cgatep)

                # logical_effort

                g_nor2 = (stack_2p * pnrat + 1) / (1 + pnrat)
                g_nand2 = (stack_2n + pnrat) / (1 + pnrat)

                # for nonoverlap driver the ratio of the out1 load cap mutliplied by that stages logical effort to the load cap of out2 times
                # that stages logical effort (the logical effort of the stage that is being driven)
                # cload_ratio=(C2/C1)*(g_nor2)/(g_nand2)
                # cload_ratio_w_M5 = ((3.0*C2)/(2.0*C1))*(g_nor2)/(g_nand2)

                g_flying_inv = stack_3p * pnrat / (1 + pnrat)

                # g_nov = nov_driver.get_logical_effort(cload_ratio=cload_ratio)#((1+pnrat)*(nov_ratio*2*nov_pg_ratio+1))/(1+pnrat)#assume that the

                g_nov_out2 = (1 + stack_2p * pnrat + stack_2p * pnrat * (gamma
                              * 2.0 / delay_fanout + 4.0 / delay_fanout ** 3)) \
                    / (1 + pnrat)
                g_nov_out1 = (stack_2n + pnrat + stack_2n * (gamma * 2.0
                              / delay_fanout + 4.0 / delay_fanout ** 3)) / (1
                        + pnrat)
                g_pg = 1  # i am sizing the pg like an inverter and sizing the inverter driving the pmos so Cin is the same as an inverter and ignoring a bunch of other stuff

                # one of the non-overlap drivers drives 3 gates instead of 2 so the opitmization should be done for that

                # C_load_M14_driver = C1+C4
                # C_load_M23_driver = C2+C3
                # C_load_w_M5 = C5 + C1 + C2 + C3 + C4
                # C_load_wo_M5 = C1 + C2 + C3 + C4
                # g6_w_M5 = nov_driver.get_logical_effort(cload_ratio=cload_ratio_w_M5)

                # g6_wo_M5 = nov_driver.get_logical_effort(cload_ratio=cload_ratio)
                # branching_5 = C_load_wo_M5*g6_wo_M5/ \
                #             (C_load_w_M5*g6_w_M5)

                # 1 clk2_h path calculate expression for equivalent branching factor given fixed C loads of the switches

                b8 = 1  # includes driver of M5
                b7 = 2  # nonoverlap driver
                b6 = 1 + g_nor2 * g_nov_out1 * C2 / (g_nand2 * g_nov_out2 * C1)
                b5 = 2 + 0.1  # 0.1 for bottom plate parasitics
                b4 = 1
                b3 = 25 / 9.0  # set by phase splitter; need to work out actual number
                b2 = 1
                b1 = 1

                # b0 = 1 #inverter inside the comparator

                print 'Branching factors: %s   %s   %s   %s   %s   %s   %s' % (
                    b1,
                    b2,
                    b3,
                    b4,
                    b5,
                    b6,
                    b7,
                    )

                # 2 calculate optimal fanout per stage

                g1 = 1  # inverter inside comparator (second stage with fixed size)
                g2 = g_pg  # pass gate in d flop
                g3 = 1  #
                g4 = 1  # stage 1 of phase splitter
                g5 = 1  # stage 2 of phase splitter
                g6 = 1  # level shift inverter
                g7 = g_nov_out2
                g8 = g_nand2

                cin1 = oop.control_core.comparator.Wn.value \
                    * oop.control_core.comparator.nmult_inv2.value * cgaten \
                    + oop.control_core.comparator.Wp.value \
                    * oop.control_core.comparator.pmult_inv2.value * cgatep

                # cin1 = 0.4e-6*(cgaten+pnrat*cgatep)    #(minimum sized inverter (with M1 min area constraint))

                G = g1 * g2 * g3 * g4 * g5 * g6 * g7 * g8
                F = C1 / cin1
                B = b1 * b2 * b3 * b4 * b5 * b6 * b7 * b8
                PE = G * F * B
                h = PE ** (1 / 8.0)

                if target_effort:
                    h = target_effort
                self.target_effort = h

                # now use this to size the path
                # size the M1 and M6 nand gates
                # pdb.set_trace()

                for driver in (oop.swcap_w_drivers_in_phase.M1_driver,
                               oop.swcap_w_drivers_in_phase.M6_driver,
                               oop.swcap_w_driver_out_phase.M1_driver,
                               oop.swcap_w_driver_out_phase.M6_driver):
                    cin8_nand = driver.size_for_effort(
                        pnrat=pnrat,
                        stack_2n=stack_2n,
                        effort=h,
                        cload=C1,
                        branching=1,
                        cgate=cgaten,
                        )
                    driver.set_parameters('device_intent',
                                          later_stage_device_intent)

                # size M4 and M9 nor gates and M5 driver
                # for now just size with the same fanout

                for driver in (oop.swcap_w_drivers_in_phase.M4_driver,
                               oop.swcap_w_drivers_in_phase.M9_driver,
                               oop.swcap_w_driver_out_phase.M4_driver,
                               oop.swcap_w_driver_out_phase.M9_driver):
                    driver.size_for_effort(
                        pnrat=pnrat,
                        stack_2n=stack_2n,
                        effort=h,
                        cload=C4,
                        branching=1,
                        cgate=cgaten,
                        )
                    driver.set_parameters('device_intent',
                                          later_stage_device_intent)

                # size M3 and M8 nor gates and M5 driver
                # for now just size with the same fanout

                for driver in (oop.swcap_w_drivers_in_phase.M3_driver,
                               oop.swcap_w_drivers_in_phase.M8_driver,
                               oop.swcap_w_driver_out_phase.M3_driver,
                               oop.swcap_w_driver_out_phase.M8_driver):
                    cin8_nor = driver.size_for_effort(
                        pnrat=pnrat,
                        stack_2p=stack_2p,
                        effort=h,
                        cload=C3,
                        branching=1,
                        cgate=cgatep,
                        )
                    driver.set_parameters('device_intent',
                                          later_stage_device_intent)

                print '================='

                # size M2 and M7 nor gates and M5 driver
                # for now just size with the same fanout

                for driver in (oop.swcap_w_drivers_in_phase.M2_driver,
                               oop.swcap_w_drivers_in_phase.M7_driver,
                               oop.swcap_w_driver_out_phase.M2_driver,
                               oop.swcap_w_driver_out_phase.M7_driver):
                    driver.size_for_effort(
                        pnrat=pnrat,
                        stack_2p=stack_2p,
                        effort=h,
                        cload=C2,
                        branching=1,
                        cgate=cgatep,
                        )
                    driver.set_parameters('device_intent',
                                          later_stage_device_intent)
                print '========sfsdfsdfsf========='

                for driver in (oop.swcap_w_drivers_in_phase.M5_driver,
                               oop.swcap_w_drivers_in_phase.M5_driver):
                    driver.size_for_effort(
                        pnrat=pnrat,
                        stack_3p=stack_3p,
                        effort=h,
                        cload=C5,
                        branching=1,
                        cgate=cgatep,
                        )
                    driver.set_parameters('device_intent',
                                          later_stage_device_intent)
```

```

                   # need to size paths to other gates for the same delay
                   # pdb.set_trace()
                   # size the clock driver
                   # cp = oop.control_core.clock_driver

                   # size the test circuit
                   # pdb.set_trace()
                   # now size the nov level shifters
                   # these return cin6 sideload because stage 6 and 7 are in this block

                   cp = oop.control_core.critical_path
                   (cin6, cin6_sideload) = cp.nov_ls.nov_high.size_for_effort(
                       effort=h,
                       branching=b7,
                       branching2=b6,
                       cload=cin8_nand,
                       cload_ratio=cin8_nor / cin8_nand,
                       device_intent=later_stage_device_intent,
                       )
                   (cin6, cin6_sideload) = cp.nov_ls.nov_low.size_for_effort(
                       effort=h,
                       branching=b7,
                       branching2=b6,
                       cload=cin8_nand,
                       cload_ratio=cin8_nor / cin8_nand,
                       device_intent=later_stage_device_intent,
                       )
                   (cin6, cin6_sideload) = cp.nov_ls_b.nov_high.size_for_effort(
                       effort=h,
                       branching=b7,
                       branching2=b6,
                       cload=cin8_nand,
                       cload_ratio=cin8_nor / cin8_nand,
                       device_intent=later_stage_device_intent,
                       )
                   (cin6, cin6_sideload) = cp.nov_ls_b.nov_low.size_for_effort(
                       effort=h,
                       branching=b7,
                       branching2=b6,
                       cload=cin8_nand,
                       cload_ratio=cin8_nor / cin8_nand,
                       device_intent=later_stage_device_intent,
                       )

                   # now size the level shifter caps (at inputs and outputs)
                   # input caps first; they are 10 times the size of the input capacitance
    they are driving.  This is accomplished
                   # by setting the channel length 10 times larger than normal.

                   max_width = 20e-6
                   W_ls_cap = snap(cin6 / cgaten)
                   mult = 1.0
                   while W_ls_cap > max_width:
                       W_ls_cap = snap(W_ls_cap * mult / (mult + 1))
                       mult = mult + 1.0
                   for ls_cap in (oop.control_core.cap_ls_1,
                                  oop.control_core.cap_ls_2):
                       ls_cap.W = W_ls_cap
                       ls_cap.L = snap(L * 10.0)
                       ls_cap.mult = mult * 2  # times 2 because we are using thick ox
```

```python
            ls_cap.device_intent = 'high_voltage'
        print 'Sizing level shifter caps to W=%s L=%s mult=%s' \
            % (ls_cap.W.value, ls_cap.L.value, ls_cap.mult.value)

        # now output caps
        # Ccoupling is 1/4 the size of Cload

        W_coupling = snap(cin8_nand / cgaten)
        mult = 1.0
        while W_coupling > max_width:
            W_coupling = snap(W_coupling * mult / (mult + 1))
            mult = mult + 1.0
        for nov_cap in (cp.M_clk1_coupling, cp.M_clk2_coupling,
                        cp.M_clk1b_coupling, cp.M_clk2b_coupling):
            nov_cap.L = L
            nov_cap.mult = mult * 2  # times 2 because we are using thick ox
            nov_cap.W = W_coupling
            nov_cap.device_intent = 'high_voltage'
            if snap(W_coupling / 10.0) > Wmin:
                nov_cap.W = snap(W_coupling / 10.0)
                nov_cap.L = snap(L * 10.0)

        print 'Sizing gate driver coupling caps to W=%s L=%s mult=%s' \
            % (nov_cap.W.value, nov_cap.L.value, nov_cap.mult.value)

        print 'Target effort: %s' % self.target_effort

        # now size the phase_splitter

        phase_splitter_load_cap = cp.phase_splitter.size_for_effort(
            pnrat=pnrat,
            effort=h,
            cload=cin6,
            branching=b5,
            cgate=cgaten,
            device_intent=later_stage_device_intent,
            )

        # now size the dflop

        # import pdb
        # pdb.set_trace()

        dflop_load_cap = cp.dflop.size_for_effort(
            pnrat=pnrat,
            effort=h,
            cload=phase_splitter_load_cap,
            branching=b3,
            cgate=cgaten,
            device_intent=later_stage_device_intent,
            )

        #

        ccell = oop.control_core

        # now the clk_output driver has been sized, we can size the delay cell

        cload_clk_driver = cgaten \
            * (ccell.clk_driver.toggle_out_nor.Wn.value
               * ccell.clk_driver.toggle_out_nor.nmult.value
```

```
                        + ccell.clk_driver.toggle_out_nor.Wp.value
                        * ccell.clk_driver.toggle_out_nor.pmult.value
                        + ccell.clk_driver.clk_driver.M1.W.value
                        * ccell.clk_driver.clk_driver.M1.mult.value
                        + ccell.clk_driver.clk_driver.NM3.W.value
                        * ccell.clk_driver.clk_driver.NM3.mult.value)
               ccell.toggle_clock_delay.size_for_effort(cload=cload_clk_driver)

               # ccell.core1.cfly = 10e-9/18.0
               # ccell.core2.cfly = 10e-9/18.0
               # ccell.core1.rsw_on = 0.02
               # ccell.core2.rsw_on = 0.02
               # ccell.core1.cfly_ic = vref
               # ccell.core2.cfly_ic = vref

               # size volt_position_control

               switch_m2_width = oop.swcap_w_drivers_in_phase.I0.I0.M2.W.value \
                   * oop.swcap_w_drivers_in_phase.I0.I0.M2.mult.value
               ccell.volt_position_control.simple_size(width=80.0e-6)  # hard code this for now

               # pdb.set_trace()
               # size the inverter inside the comparator if needed
               # need to extract the sideloads and redo calculations with sideloads accounted for

               print 'Win_7 = %s       Win_6 = %s             Win_4 = %s' \
                   % (cin8_nand / cgate, cin6 / cgate, phase_splitter_load_cap
                      / cgate)
               oop.load_dac.bits = 7
               print 'M1_driver is driving a total width of %s and has a logical effort of %s' \
                   % (C1 / cgaten, g8)
               print 'M1_driver input width is %s' % (cin8_nand / cgate)
               print 'Nonoverlap is driving a total width of %s and has a logical_effort of %s' \
                   % (b7 * cin8_nand / cgate, g7)
               print 'Nonoverlap input width is %s' % (cin6 / cgate)
               print 'Phase Interp Stage2 is driving a total width of %s and has a logical_effort of %s' \
                   % (b5 * cin6 / cgate, g5)
               phis2 = cp.phase_splitter.buf_stage_2.Wp.value \
                   + cp.phase_splitter.buf_stage_2.Wn.value
               print 'Phase Interp Stage2 input width is %s' % phis2
               print 'Phase Interp Stage1 is driving a total width of %s and has a logical_effort of %s' \
                   % (phis2, g4)
               phis1 = cp.phase_splitter.inverting_stage.Wp.value \
                   + cp.phase_splitter.inverting_stage.Wn.value
               print 'Phase Interp Stage1 input width is %s' \
                   % (phase_splitter_load_cap / cgate)
               print 'Dflop buffer is driving a total width of %s and has a logical_effort of %s' \
                   % (phase_splitter_load_cap / cgate, g3)
               dflop_inv = cp.dflop.output_driver_Wn.value \
                   * cp.dflop.output_driver_nmult.value * (1 + pnrat)
               print 'Dflop buffer input width is %s' % dflop_inv
               print 'Dflop pg is driving a total width of %s and has a logical_effort of %s' \
                   % (dflop_inv, g2)
```

```

                # dflop_inv = cp.dflop.output_driver_Wn.value*(1+pnrat)

                print 'Dflop pg input width is %s' % (dflop_load_cap / cgate)
                print 'comparator gate should be driving a Dflop pg input of %s' \
                    % (cin1 / cgate * self.target_effort)

                # pdb.set_trace()

                self.fix_widths()
```

A.2 BAG PyCell Code

This section contains PyCell code for examples of several different layout styles.

A.2.1 5 Transistor Amplifier PyCell

```
#!/usr/bin/python
# -*- coding: utf-8 -*-
from __future__ import with_statement

__version__ = '$Revision: #1 $'
__author__ = 'John Crossley'

from BAG_layout import *
import pdb
from operator import itemgetter
import inspect


class simple_amp(std_cell):

    '''''
    '''

    paramNames = dict(L='L', Wn='Wn', Wp='Wp', Wtail='Wtail',
                      tail_height_ratio='tail_height_ratio')

    # types: 'rail', 'route_group' (NORTH to SOUTH list of ordered groups), route, devi
ce_row, device

    layout_info = [  # shared_source, shared_drain, series, parallel
                     # 'enforce_even_fingers' : True,   #
        {
            'name': 'VDD',
            'type': 'rail',
            'layer': 'metal1',
            'terminal': 'VDD',
            'term_type': 'INPUT_OUTPUT',
            'contact_to': 'diffusion',
            'enclose_contact': ['nimplant', 'nwell'],
            },
        {
            'name': 'p_row1',
            'type': 'device_group',
            'tran_type': 'pmos',
            'height_ratio': None,
            'vertical_align': 'SOUTH',
            'subelements': [{
                'name': 'Mpmos',
                'param_mapping': {'W': 'Wp', 'L': 'L'},
                'G': ['OUTN', 'OUTN'],
                'D': ['OUTN', 'OUTP'],
                'S': ['VDD'],
                'B': ['VDD'],
                'layout_type': 'shared_source',
                'align_position': 'CENTER',
                'align_point': 1,
                }],
            },
        {'name': 'route_group1', 'type': 'route_group',
         'subelements': [{
            'name': 'OUTN',
            'layer': 'metal1',
            'terminal': None,
            'term_type': None,
            }, {
            'name': 'OUTP',
            'layer': 'metal1',
            'terminal': 'OUTP',
            'term_type': 'OUTPUT',
            }]},
        {
            'name': 'n_row1',
            'type': 'device_group',
            'tran_type': 'nmos',
            'height_ratio': None,
            'vertical_align': 'NORTH',
            'subelements': [{
                'name': 'Mnmos',
                'param_mapping': {'W': 'Wn', 'L': 'L'},
                'G': ['INP', 'INN'],
                'D': ['OUTN', 'OUTP'],
                'S': ['source'],
                'B': ['GND'],
                'layout_type': 'shared_source',
                'align_position': 'CENTER',
                'align_point': 1,
                }],
            },
        {'name': 'route_group2', 'type': 'route_group',
         'subelements': [{
            'name': 'INN',
            'layer': 'metal1',
            'terminal': 'INN',
            'term_type': 'INPUT',
            }, {
            'name': 'INP',
            'layer': 'metal1',
            'terminal': 'INP',
            'term_type': 'INPUT',
            }, {
            'name': 'source',
            'layer': 'metal1',
            'terminal': None,
            'term_type': None,
            }, {
                   'name': 'BIAS',
                   'layer': 'metal1',
                   'terminal': 'BIAS',
            'term_type': 'INPUT',
            }]},
        {
            'name': 'n_row2',
            'type': 'device_group',
            'tran_type': 'nmos',
            'height_ratio': 'tail_height_ratio',
            'vertical_align': 'NORTH',
            'subelements': [{
                'name': 'Mtail',
                'param_mapping': {'W': 'Wtail', 'L': 'L'},
                'G': ['BIAS'],
                'D': ['source'],
                'S': ['GND'],
                'B': ['GND'],
                'layout_type': 'single',
                'align_position': 'CENTER',
                'align_point': 1,
                }],
            },
        {
            'name': 'GND',
            'type': 'rail',
            'layer': 'metal1',
            'terminal': 'GND',
            'term_type': 'INPUT_OUTPUT',
            'contact_to': 'diffusion',
            'enclose_contact': ['pimplant'],
            },
        ]

    default = dict(Wn=1.0, nmos_ratio=0.64, verticalPitch=10.2,
                   extra_rail_space_bottom=0.12)

     # ####################################################################

    @classmethod
    def defineParamSpecs(cls, specs):
        """Define the PyCell parameters.  The order of invocation of
        specs() becomes the order on the form.

        Arguments:
        specs - (ParamSpecArray)  PyCell parameters

        These are the parameters in the paramNames dictionary - define them here
            """

        [selfcopy, hlink] = printmeth(cls)

        # Super makes all the parent functions available in the new defineParamSpecs

        super(simple_amp, cls).defineParamSpecs(specs)
        mySpecs = ParamSpecArray()

        # Set up specs

        L = specs.tech.getMosfetParams('nmos', cls.oxide, 'minLength')  # minimum length (get from tech file)
        L = cls.grid.snap(L)  # snap to grid

        # stepConstraint = starting value, step size, and resolution (maybe number of points?) or max value

        stepConstraint = StepConstraint(cls.maskgrid, start=L,
                resolution=cls.resolution, action=FailAction.REJECT)

        # myspecs: (name, (default OR L you've defined), constraint)

        mySpecs('L', cls.default.get('L', L), constraint=stepConstraint)

        Wn = specs.tech.getMosfetParams('nmos', cls.oxide, 'minWidth')
        Wn = cls.grid.snap(Wn)
        stepConstraint = StepConstraint(cls.maskgrid, start=Wn,
                resolution=cls.resolution, action=FailAction.REJECT)
        mySpecs('Wn', cls.default.get('Wn', Wn),
                constraint=stepConstraint)
        mySpecs('Wp', cls.default.get('Wp', Wn),
                constraint=stepConstraint)
        mySpecs('Wtail', cls.default.get('Wtail', Wn),
                constraint=stepConstraint)
        mySpecs('tail_height_ratio', 0.25)

        # Parameter renaming: adds mySpecs (what you've defined above) into specs

        renameParams(mySpecs, specs, cls.paramNames)

        # This is a Debugging call and should ALWAYS be the last line of a method

        savemeth(cls, selfcopy, hlink)

    # ###################################################################

    def define_derived_parameters(self):
        '''
        Any derived parameters that are referenced in self.layout_info should be defined here

        For example, in the pmos device_group below the param_mapping of 'W' is 'Wp'.  If Wp is not a
        parameter defined at the pycell level, then self.Wp must be defined in this function.

                          'subelements' : [  {  'name':'M3M4',
                                                'param_mapping': {'W':'Wp' },
                                                'G' : ['IN1','IN2'],
                                                'D' : ['OUT'],
                                                'S' : ['VDD'],
                                                'B' : ['VDD'],
                                                'layout_type' : 'series'     #shared_source, shared_drain, series, parallel
                                             },

                  def define_derived_parameters(self) :
                      self.Wp = self.grid.snap(self.PNratio*self.Wn)

        Basically, the contents of params get turned into fields, so here you should add any derived parameters as fields if they
```

```
                don't appear in params
                '''

                [selfcopy, hlink] = printmeth(self)

                # This is a Debugging call and should ALWAYS be the last line of a method

                savemeth(self, selfcopy, hlink)
```

A.2.2 Asynchronous Comparator

```
#!/usr/bin/python
# -*- coding: utf-8 -*-
from __future__ import with_statement

__version__ = '$Revision: #1 $'
__author__ = 'John Crossley'

from BAG_layout import *
import pdb
from operator import itemgetter


class async_comp(std_cell_row):

    '''''
    This is a new version of the async_comp that inherits from std_cell_row
    '''

    paramNames = dict(
        L='L',
        input_width='input_width',
        mult_input_mirror='mult_input_mirror',
        bias_enable_width='bias_enable_width',
        mult_bias_enable_mirror='mult_bias_enable_mirror',
        bias_width='bias_width',
        mult_input='mult_input',
        mult_precharge='mult_precharge',
        mult_bias_mirror='mult_bias_mirror',
        mult_bias_enable='mult_bias_enable',
        mult_bias='mult_bias',
        inv_feedback_width='inv_feedback_width',
        mult_feedback='mult_feedback',
        inv_n_width='inv_n_width',
        inv_p_width='inv_p_width',
        inv2_n_width='inv2_n_width',
        inv2_p_width='inv2_p_width',
        inv_en_n_width='inv_en_n_width',
        inv_en_p_width='inv_en_p_width',
        extra_vertical_pitch_vref='extra_vertical_pitch_vref',
        decap_p_width='decap_p_width',
        decap_length='decap_length',
        decap_n_width='decap_n_width',
        decap_n_type='decap_n_type',
        decap_p_type='decap_p_type',
        )

    # types:  'route_group' (WEST to EAST list of ordered routes), route, std_cell, std_cell_row

    pin_info = [{
        'name': 'VREF',
        'term_type': 'INPUT_OUTPUT',
        'subelements': [{'instance': 'async_comp_p1', 'terminal': 'VDD'
                        }, {'instance': 'M_bias_decouple',
                        'terminal': 'VDD'},
                        {'instance': 'M_bias_decouple_VREF_GND',
                        'terminal': 'VDD'}],
        'type': 'rail',
        }, {
        'name': 'VDD',
        'term_type': 'INPUT_OUTPUT',
        'subelements': [{'instance': 'async_comp_p2', 'terminal': 'VDD'
                        }, {'instance': 'async_comp_p3',
                        'terminal': 'VDD'}],
        'type': 'rail',
        }, {
        'name': 'GND',
        'term_type': 'INPUT_OUTPUT',
        'subelements': [{'instance': 'M_bias_decouple_VREF_GND',
                        'terminal': 'GND'},
                        {'instance': 'M_bias_decouple',
                        'terminal': 'GND'}, {'instance': 'async_comp_p1'
                        , 'terminal': 'GND'},
                        {'instance': 'async_comp_p2', 'terminal': 'GND'
                        }, {'instance': 'async_comp_p3',
                        'terminal': 'GND'}],
        'type': 'rail',
        }]

    layout_info = [  # extra space on left and right of this routing group (in um)
                     # {'name': 'con_vref', 'layer':'metal', 'terminal': None, 'term_type': None },
                     # extra space on left and right of this routing group (in um)
                     #
                     # extra space on left and right of this routing group (in um)
                     # {'name': 'int', 'layer':'metal2', 'terminal': None, 'term_type': None },
                     # extra space on left and right of this routing group (in um)
                     # 'mirror_y' : 'True',
                     # extra space on left and right of this routing group (in um)
                     # 'mirror_y' : 'True',
                     # extra space on left and right of this routing group (in um)
                     # extra space on left and right of this routing group (in um)
        {
            'name': 'M_bias_decouple_VREF_GND',
            'type': 'std_cell',
            'lib_cell': 'BAG_std_cells_oa/cap_decouple',
            'param_mapping': {
                'Wn': 'decap_n_width',
                'Wp': 'decap_p_width',
                'L': 'decap_length',
                'extra_vertical_pitch_top': 'extra_vertical_pitch_vref'
                    ,
                    'verticalPitch': 'p1_verticalPitch',
                    'extend_enclosing_layers_to_pr': 'extend_enclosing_layers_pmos_cap'
                    ,
                    'device_type_nmos': 'decap_n_type',
                    'device_type_pmos': 'decap_p_type',
                    'extra_space_for_thick_ox': 'extra_space_for_thick_ox',
                    },
                'pin_mapping': {'VDD': 'con_vref', 'GND': 'GND'},
                },
            {
                'name': 'route_groupm1',
                'type': 'route_group',
                'extra_space': 0.4,
                'subelements': [{
                    'name': 'bias',
                    'layer': 'metal2',
                    'terminal': None,
                    'term_type': None,
                    }],
                },
            {
                'name': 'M_bias_decouple',
                'type': 'std_cell',
                'lib_cell': 'BAG_std_cells_oa/async_cap_decouple',
                'param_mapping': {
                    'Wn': 'decap_n_width',
                    'Wp': 'decap_p_width',
                    'L': 'decap_length',
                    'extra_vertical_pitch_top': 'extra_vertical_pitch_vref'
                        ,
                    'verticalPitch': 'p1_verticalPitch',
                    'extend_enclosing_layers_to_pr': 'extend_enclosing_layers_pmos_c
ap'
                        ,
                    'device_type_nmos': 'decap_n_type',
                    'device_type_pmos': 'decap_p_type',
                    'extra_space_for_thick_ox': 'extra_space_for_thick_ox',
                    },
                'pin_mapping': {
                    'VDD': 'con_vref',
                    'comp_bias': 'bias',
                    'comp_bias_global': 'bias2',
                    'GND': 'GND',
                    },
                },
            {
                'name': 'route_group0',
                'type': 'route_group',
                'extra_space': 0.2,
                'subelements': [{
                    'name': 'bias',
                    'layer': 'metal2',
                    'terminal': None,
                    'term_type': None,
                    }, {
                    'name': 'bias2',
                    'layer': 'metal2',
                    'terminal': None,
                    'term_type': None,
                    }, {
                    'name': 'con_vref',
                    'layer': 'metal1',
                    'terminal': None,
                    'term_type': None,
                    }],
                },
            {
                'name': 'async_comp_p1',
                'type': 'std_cell',
                'lib_cell': 'BAG_std_cells_oa/async_comp_p1',
                'param_mapping': {
                    'input_width': 'input_width',
                    'mult_input_mirror': 'mult_input_mirror',
                    'bias_enable_width': 'bias_enable_width',
                    'mult_bias_enable_mirror': 'mult_bias_enable_mirror',
                    'bias_width': 'bias_width',
                    'mult_bias_mirror': 'mult_bias_mirror',
                    'extra_vertical_pitch_top': 'extra_vertical_pitch_vref'
                        ,
                    'verticalPitch': 'p1_verticalPitch',
                    'extend_enclosing_layers_to_pr': 'extend_enclosing_layers_pmos_cap'
                        ,
                    },
                'pin_mapping': {
                    'bias': 'bias',
                    'bias2': 'bias2',
                    'vref_in': 'con_vref',
                    'VDD': 'con_vref',
                    'GND': 'GND',
                    },
                },
            {
                'name': 'route_group1',
                'type': 'route_group',
                'extra_space': 0.2,
                'subelements': [{
                    'name': 'bias',
                    'layer': 'metal2',
                    'terminal': 'bias',
                    'term_type': 'INPUT_OUTPUT',
                    'extra_space': 0.0,
                    }, {
                    'name': 'bias2',
                    'layer': 'metal2',
                    'terminal': 'bias2',
                    'term_type': 'INPUT_OUTPUT',
                    'extra_space': 0.0,
                    }],
                },
            {
                'name': 'async_comp_p2',
                'type': 'std_cell',
                'lib_cell': 'BAG_std_cells_oa/async_comp_p2',
                'param_mapping': {
                    'input_width': 'input_width',
                    'mult_input': 'mult_input',
                    'mult_precharge': 'mult_precharge',
                    'bias_enable_width': 'bias_enable_width',
                    'mult_bias_enable': 'mult_bias_enable',
                    'bias_width': 'bias_width',
                    'mult_bias': 'mult_bias',
                    'share_nwell_left': 'share_nwell_left_p2',
                    'extend_enclosing_layers_to_pr': 'extend_enclosing_layers_to_pr_p2'
                        ,
                    },
                'pin_mapping': {
                    'bias': 'bias',
                    'bias2': 'bias2',
                    'VDD': 'VDD',
                    'GND': 'GND',
                    'enable': 'int_enable',
                    'int': 'int',
                    },
                },
            {
                'name': 'route_group10',
                'type': 'route_group',
                'extra_space': 0.2,
                'subelements': [{
                    'name': 'int',
                    'layer': 'metal2',
                    'terminal': None,
                    'term_type': None,
                    'extra_space': 0.1,
                    }, {
                    'name': 'int_enable',
                    'layer': 'metal2',
                    'terminal': 'int_enable',
                    'term_type': 'INPUT_OUTPUT',
                    }],
                },
            {
                'name': 'inv_enable_2',
                'type': 'std_cell',
                'lib_cell': 'BAG_std_cells_oa/Inverter_pass_through',
                'param_mapping': {'L': 'L', 'Wn': 'Wn_inv',
                                  'PNratio': 'pnrat_inv'},
                'pin_mapping': {
                    'IN': 'enable_b',
                    'OUT': 'int_enable',
                    'int': 'int',
                    'VDD': 'VDD',
                    'GND': 'GND',
                    },
                },
            {
                'name': 'route_group11',
                'type': 'route_group',
                'extra_space': 0.2,
                'subelements': [{
                    'name': 'enable_b',
                    'layer': 'metal2',
                    'terminal': None,
                    'term_type': None,
                    }, {
                    'name': 'int',
                    'layer': 'metal2',
                    'terminal': None,
                    'term_type': None,
                    'extra_space': 0.1,
                    }],
                },
            {
                'name': 'inv_enable_1',
                'type': 'std_cell',
                'lib_cell': 'BAG_std_cells_oa/Inverter_pass_through',
                    'param_mapping': {'L': 'L', 'Wn': 'Wn_inv',
                                      'PNratio': 'pnrat_inv'},
                    'pin_mapping': {
                        'IN': 'enable',
                        'OUT': 'enable_b',
                        'int': 'int',
                        'VDD': 'VDD',
                        'GND': 'GND',
                        },
                    },
                {
                    'name': 'route_group12',
                    'type': 'route_group',
                    'extra_space': 0.2,
                    'subelements': [{
                        'name': 'enable',
                        'layer': 'metal2',
                        'terminal': 'enable',
                        'term_type': 'INPUT',
                        }, {
                        'name': 'int',
                        'layer': 'metal2',
                        'terminal': 'internal_node',
                        'term_type': 'INPUT_OUTPUT',
                        'extra_space': 0.1,
                        }],
                    },
                {
                    'name': 'async_comp_p3',
                    'type': 'std_cell',
                    'lib_cell': 'BAG_std_cells_oa/async_comp_p3_v2',
                    'param_mapping': {
                        'inv_feedback_width': 'inv_feedback_width',
                        'mult_feedback': 'mult_feedback',
                        'inv_n_width': 'inv_n_width',
                        'inv_p_width': 'inv_p_width',
                        'inv2_n_width': 'inv2_n_width',
                        'inv2_p_width': 'inv2_p_width',
                        },
                    'pin_mapping': {
                        'VDD': 'VDD',
                        'GND': 'GND',
                        'out': 'out',
                        'in': 'int',
                        },
                    },
                {
                    'name': 'route_group_out',
                    'type': 'route_group',
                    'extra_space': 0.2,
                    'subelements': [{
                        'name': 'out',
                        'layer': 'metal2',
                        'terminal': 'out',
                        'term_type': 'OUTPUT',
                        }],
                    },
                ]

            default = dict(Wn=0.4, nmos_ratio=0.45, PNRatio=2.0,
                           verticalPitch=5.0, railWidth=0.36)
```

```

     # ###################################################################

    @classmethod
    def defineParamSpecs(cls, specs):
        """Define the PyCell parameters.  The order of invocation of
        specs() becomes the order on the form.

        Arguments:
        specs - (ParamSpecArray)  PyCell parameters
            """

        [selfcopy, hlink] = printmeth(cls)
        super(async_comp, cls).defineParamSpecs(specs)
        mySpecs = ParamSpecArray()

        Wn = specs.tech.getMosfetParams('nmos', cls.oxide, 'minWidth')
        Wn = cls.grid.snap(Wn)
        stepConstraint = StepConstraint(cls.maskgrid, start=Wn,
                resolution=cls.resolution, action=FailAction.REJECT)

        Wn = cls.default.get('Wn', Wn)
        Wn = cls.grid.snap(Wn)

        L = specs.tech.getMosfetParams('nmos', cls.oxide, 'minLength')
        L = cls.grid.snap(L)
        stepConstraint = StepConstraint(cls.maskgrid, start=L,
                resolution=cls.resolution, action=FailAction.REJECT)
        mySpecs('L', cls.default.get('L', L), constraint=stepConstraint)

        # "" choice for type lets the lower level transistor wrapper choose the default

        nmos_types = cls.tech_yaml['devices']['nmos4']['device_types'
                ].keys() + ['']
        mySpecs('decap_n_type', '',
                constraint=ChoiceConstraint(nmos_types, REJECT))
        pmos_types = cls.tech_yaml['devices']['pmos4']['device_types'
                ].keys() + ['']
        mySpecs('decap_p_type', '',
                constraint=ChoiceConstraint(pmos_types, REJECT))

        mySpecs('input_width', cls.default.get('input_width', Wn),
                constraint=stepConstraint)
        mySpecs('bias_enable_width', cls.default.get('bias_enable_width'
                , Wn), constraint=stepConstraint)
        mySpecs('bias_width', cls.default.get('bias_width', Wn),
                constraint=stepConstraint)
        mySpecs('inv_feedback_width',
                cls.default.get('inv_feedback_width', Wn),
                constraint=stepConstraint)
        mySpecs('inv_n_width', cls.default.get('inv_n_width', Wn),
                constraint=stepConstraint)
        mySpecs('inv_p_width', cls.default.get('inv_p_width', Wn),
                constraint=stepConstraint)
        mySpecs('inv2_n_width', cls.default.get('inv2_n_width', Wn),
                constraint=stepConstraint)
        mySpecs('inv2_p_width', cls.default.get('inv2_p_width', Wn),
                constraint=stepConstraint)
        mySpecs('inv_en_n_width', cls.default.get('inv_en_n_width',
                Wn), constraint=stepConstraint)
                mySpecs('inv_en_p_width', cls.default.get('inv_en_p_width',
                        Wn), constraint=stepConstraint)
                mySpecs('decap_n_width', cls.default.get('decap_n_width', 10
                        * Wn), constraint=stepConstraint)
                mySpecs('decap_p_width', cls.default.get('decap_p_width', 10
                        * Wn), constraint=stepConstraint)
                mySpecs('decap_length', cls.default.get('decap_length', 10
                        * L), constraint=stepConstraint)

                mySpecs('mult_feedback', cls.default.get('mult_feedback', 2),
                        constraint=StepConstraint(1, start=1,
                        action=FailAction.REJECT))
                mySpecs('mult_input_mirror', cls.default.get('mult_input_mirror'
                        , 2), constraint=StepConstraint(1, start=1,
                        action=FailAction.REJECT))
                mySpecs('mult_bias_enable_mirror',
                        cls.default.get('mult_bias_enable_mirror', 2),
                        constraint=StepConstraint(1, start=1,
                        action=FailAction.REJECT))
                mySpecs('mult_bias_mirror', cls.default.get('mult_bias_mirror',
                        2), constraint=StepConstraint(1, start=1,
                        action=FailAction.REJECT))
                mySpecs('mult_input', cls.default.get('mult_input', 2),
                        constraint=StepConstraint(1, start=1,
                        action=FailAction.REJECT))
                mySpecs('mult_precharge', cls.default.get('mult_precharge', 2),
                        constraint=StepConstraint(1, start=1,
                        action=FailAction.REJECT))
                mySpecs('mult_bias_enable', cls.default.get('mult_bias_enable',
                        2), constraint=StepConstraint(1, start=1,
                        action=FailAction.REJECT))
                mySpecs('mult_bias', cls.default.get('mult_bias', 2),
                        constraint=StepConstraint(1, start=1,
                        action=FailAction.REJECT))
                mySpecs('extra_vertical_pitch_vref', 0.0,
                        constraint=RangeConstraint(0.0, None,
                        action=FailAction.REJECT))

                # Parameter renaming

                renameParams(mySpecs, specs, cls.paramNames)

                # This is a Debugging call and should ALWAYS be the last line of a metho
    d

                savemeth(cls, selfcopy, hlink)

            # ##################################################################

            def define_derived_parameters(self):
                '''
                Any derived parameters that are referenced in self.layout_info should be
     defined here

                For example, in the pmos device_group below the param_mapping of 'W' is
    'Wp'.  If Wp is not a
                parameter defined a the pycell level, then self.Wp must be defined in th
    is function.

                                 'subelements' : [  {  'name':'M3M4',
                                                        'param_mapping': {'W':'Wp' },
                                                        'G' : ['IN1','IN2'],
                                                        'D' : ['OUT'],
                                                        'S' : ['VDD'],
                                                        'B' : ['VDD'],
                                                        'layout_type' : 'series'     #shared_source, shared_drain, series, parallel
                                                     },

                          def define_derived_parameters(self) :
                              self.Wp = self.grid.snap(self.PNratio*self.Wn)
                '''

                [selfcopy, hlink] = printmeth(self)
                self.share_nwell_left_p2 = 'False'
                self.nmos_ratio_pmos_cap = 0.24  # was 0.16 but reduced it for small cells
                self.p1_verticalPitch = self.verticalPitch \
                    - self.extra_vertical_pitch_vref
                self.extend_enclosing_layers_to_pr_p2 = 'False'
                self.extend_enclosing_layers_pmos_cap = 'False'
                self.Wn_inv = self.inv_en_n_width
                self.pnrat_inv = self.inv_en_p_width / self.inv_en_n_width
                self.extra_space_for_thick_ox = 0.4

                # This is a Debugging call and should ALWAYS be the last line of a method

                savemeth(self, selfcopy, hlink)

            def final_steps(self):
                [x.destroy() for x in self.layout_info[3]['subelements'
                 ][2]['routes_to_this_route_bar'].getComps() if isinstance(x,
                 Contact)]
```

A.2.3 Load DAC Array PyCell

```
#!/usr/bin/python
# -*- coding: utf-8 -*-
from __future__ import with_statement

import math
import pdb
import os
import yaml

# import PyCellCode_tsmcN65 as pcc

from cni.geo import *
from cni.constants import *
from cni.dlo import *
from BAG_layout.ArrayMethods import ArrayMethods

# from demo_cells.ArrayMethods import ArrayMethods

from cni.integ.common import createInstances, renameParams, \
    reverseDict, stretchHandle, Compare


class load_dac(ArrayMethods):

    """Define the PyCell class.
        """

    paramNames = dict(
        W_n='W_n',
        L_n='L_n',
        pwr_width_horiz='pwr_width_horiz',
        pwr_width_vert='pwr_width_vert',
        drain_width='drain_width',
        rc_width='rc_width',
        nmos_ratio='nmos_ratio',
        verticalPitch='verticalPitch',
        mirrorx='mirrorx',
        mirrory='mirrory',
        )

    default = dict(
        bits=4,
        mirrorx='true',
        mirrory='false',
        pwr_width_horiz=0.8,
        pwr_width_vert=0.8,
        verticalPitch=2.2,
        W_n=1.2,
        L_n=.06,
        nmos_ratio=.74,
        )

    oxide = 'thin'

    # Unit cell name

    cell_name = 'BAG_std_cells_oa/load_dac_ucell/layout'

    @classmethod
    def defineParamSpecs(cls, specs):
        """ Define the PyCell parameters.
            """

        super(load_dac, cls).defineParamSpecs(specs)
        minmetal1 = float(specs.tech.getPhysicalRule('minWidth',
                                  cls.layer['metal1']))
        maskgrid = specs.tech.getGridResolution()
        cls.specs = specs
        cls.tech = specs.tech

        cls.maskgrid = specs.tech.getGridResolution()
        cls.resolution = specs.tech.uu2dbu(1) * 10
        cls.grid = Grid(cls.maskgrid, snapType=SnapType.ROUND)

        mySpecs = ParamSpecArray()

        Wmin_nmos = specs.tech.getMosfetParams('nmos', cls.oxide,
                'minWidth')
        Wmin_pmos = specs.tech.getMosfetParams('pmos', cls.oxide,
                'minWidth')

        Lmin_nmos = specs.tech.getMosfetParams('nmos', cls.oxide,
                'minLength')
        Lmin_pmos = specs.tech.getMosfetParams('pmos', cls.oxide,
                'minLength')

        W_n = Wmin_nmos
        W_n = cls.grid.snap(W_n)
        stepConstraint = StepConstraint(cls.maskgrid, start=W_n,
                resolution=cls.resolution, action=FailAction.REJECT)
        mySpecs('W_n', cls.default.get('W_n', W_n),
                constraint=stepConstraint)

        L_n = Lmin_nmos
        L_n = cls.grid.snap(L_n)
        stepConstraint = StepConstraint(cls.maskgrid, start=L_n,
                resolution=cls.resolution, action=FailAction.REJECT)
        mySpecs('L_n', cls.default.get('L_n', L_n),
                constraint=stepConstraint)

                pwr_width_horiz = float(cls.default.get('pwr_width_horiz', -1))
                pwr_width_horiz_constraint = RangeConstraint(0.1, 10.0,
                        action=FailAction.REJECT)
                mySpecs('pwr_width_horiz', pwr_width_horiz,
                        constraint=pwr_width_horiz_constraint)

                pwr_width_vert = float(cls.default.get('pwr_width_vert', -1))
                pwr_width_vert_constraint = RangeConstraint(0.1, 10.0,
                        action=FailAction.REJECT)  # TODO: this shoudn't be hard coded
                specs('pwr_width_vert', pwr_width_vert,
                      constraint=pwr_width_vert_constraint)

                rc_width = float(cls.default.get('rc_width',
                                 cls.metal_min_width[cls.layer['metal2']]))
                rc_width_constraint = \
                    RangeConstraint(cls.metal_min_width[cls.layer['metal2']],
                                    10.0, action=FailAction.REJECT)
                specs('rc_width', rc_width, constraint=rc_width_constraint)

                drain_width = float(cls.default.get('drain_width',
                                    cls.metal_min_width[cls.layer['metal2']]))
                drain_width_constraint = \
                    RangeConstraint(cls.metal_min_width[cls.layer['metal2']],
                                    10.0, action=FailAction.REJECT)
                specs('drain_width', drain_width,
                      constraint=drain_width_constraint)

                nmos_ratio = cls.default.get('nmos_ratio', 0.42)
                mySpecs('nmos_ratio', nmos_ratio,
                        constraint=RangeConstraint(0.01, 0.99,
                        action=FailAction.REJECT))

                verticalPitch = cls.default.get('verticalPitch', 60.0
                        * minmetal1)
                mySpecs('verticalPitch', verticalPitch,
                        constraint=StepConstraint(cls.maskgrid,
                        start=minmetal1, resolution=cls.resolution,
                        action=FailAction.REJECT))

                renameParams(mySpecs, specs, cls.paramNames)

            def setupParams(self, params):
                """ Do parameter checking.
                """

                super(load_dac, self).setupParams(params)
                self.name_num = 0
                self.max_iter = 50
                self.bias_enable = False

                # self.max_dist = self.max_distance_from_bias

            def genTopology(self):
                """
                    """

                # First, we must calculate the number of extra contacts needed to make each cell
                # mirrorable.

                params = ParamArray(
                    cell_intent='dummy',
                    pwr_width_horiz=self.pwr_width_horiz,
                    rc_width=self.rc_width,
                    drain_width=self.drain_width,
                    verticalPitch=self.verticalPitch,
                    W_n=self.W_n,
                    L_n=self.L_n,
                    )
                instance_data = self.make_unit_cells_mirrorable(self.cell_name,
                        params)

                # Save the transistor parameters as well

                self.result_props['cs_fing_width'] = \
                    instance_data['M1_finger_width']
                self.result_props['cs_nf'] = instance_data['M1_num_fingers']

                # Define connectivity for "normal" cells - this will change for different designs!

                dummy_params = ParamArray(
                    cell_intent='dummy',
                    pwr_width_horiz=self.pwr_width_horiz,
                    rc_width=self.rc_width,
                    drain_width=self.drain_width,
                    verticalPitch=self.verticalPitch,
                    W_n=self.W_n,
                    L_n=self.L_n,
                    make_mirrorable='False',
                    extra_vdd_rail_contacts_right=self.extra_vdd_rail_contacts_right,
                    extra_gnd_rail_contacts_right=self.extra_gnd_rail_contacts_right,
                    extra_vdd_rail_contacts_left=self.extra_vdd_rail_contacts_left,
                    extra_gnd_rail_contacts_left=self.extra_gnd_rail_contacts_left,
                    )

                prefixes = {'dummy': 'D', 'normal_even': 'I', 'normal_odd': 'I'}

                self.createArray(dummy_params, dummy_params, prefixes)  # pass dummy_params to bias_params but bias is disabled

            def connect_up_to_power_grid(self):
                for hpin in self.horizontal_power_pins:
                    the_shape = hpin.getShapes()[0]
                    the_net = hpin.getTerm().getName()
                    m3_shape = Rect(self.layer['metal3'], the_shape.getBBox())
```

```
                m5_shape = Rect(self.layer['metal5'], the_shape.getBBox())
                for vpin in self.vertical_power_pins:
                    if vpin.getTerm().getName() == the_net:
                        vshape = vpin.getShapes()[0]
                        overlap = vshape.fgAnd(the_shape,
                                self.layer['metal1'])
                        p1 = overlap.getBBox().lowerLeft()
                        p2 = overlap.getBBox().upperRight()
                        AbutContact(
                            vshape.getLayer(),
                            m5_shape.getLayer(),
                            routeDir1=NORTH_SOUTH,
                            routeDir2=EAST_WEST,
                            point1=p1,
                            point2=p2,
                            abutDir=EAST_WEST,
                            )
                        overlap.destroy()
                hpin.removeShape(the_shape)
                hpin.addShape(m5_shape)
            for vpin in self.vertical_power_pins:
                the_shape = vpin.getShapes()[0]
                the_net = vpin.getTerm().getName()
                m6_shape = Rect(self.layer['metal6'], the_shape.getBBox())
                m7_shape = Rect(self.layer['metal7'], the_shape.getBBox())
                for hpin in self.horizontal_power_pins:
                    if hpin.getTerm().getName() == the_net:
                        hshape = hpin.getShapes()[0]
                        overlap = hshape.fgAnd(the_shape,
                                self.layer['metal1'])
                        p1 = overlap.getBBox().lowerLeft()
                        p2 = overlap.getBBox().upperRight()
                        AbutContact(
                            hshape.getLayer(),
                            m6_shape.getLayer(),
                            routeDir1=EAST_WEST,
                            routeDir2=NORTH_SOUTH,
                            point1=p1,
                            point2=p2,
                            abutDir=EAST_WEST,
                            )
                        overlap.destroy()
                p1 = m6_shape.getBBox().lowerLeft()
                p2 = m6_shape.getBBox().upperRight()
                AbutContact(
                    m7_shape.getLayer(),
                    m6_shape.getLayer(),
                    routeDir1=NORTH_SOUTH,
                    routeDir2=NORTH_SOUTH,
                    point1=p1,
                    point2=p2,
                    abutDir=NORTH_SOUTH,
                    )
                vpin.removeShape(the_shape)
                vpin.addShape(m7_shape)
            for hpin in self.horizontal_power_pins:
                hpin.destroy()
            del self.horizontal_power_pins

        def genLayout(self):
            """ Construct the PyCell geometries.
                """

            control_nets = ['en']
            power_nets = ['vdd', 'vss']
            ArrayMethods.genLayout(self)

            # create just the power pins so that we can move them up from M1/M2 to M7

            self.create_pins(
                self.layer['metal2'],
                [],
                0.0,
                [],
                power_nets,
                [],
                )
            self.connect_up_to_power_grid()

            # create the other pins

            [bars, y_top] = \
                self.connect_routing_channels(self.layer['metal3'],
                    self.layer['metal2'], control_nets)
            self.create_pins(
                self.layer['metal2'],
                bars,
                y_top,
                control_nets,
                [],
                [],
                )
            num_dummy_cells = 0
            num_bias_cells = 0
            for i in range(0, self.x_len):
                for j in range(0, self.y_len):
                    if self.getCellType(i, j) == 'dummy':
                        num_dummy_cells += 1
                    elif self.getCellType(i, j) == 'bias':
                        num_bias_cells += 1
            if self.result_file != '':
                self.result_props['num_dummy_cells'] = num_dummy_cells
            self.label_pins()
            self.updateProperties()

        def genInstParams(self, i, cnt_num):

            # Handle the case of the dummy inside the array, that matches with cell 0

            params = ParamArray()
            if i == 1:
                cell_type = 'dummy'
            elif cnt_num % 2 != 0:

            # Odd cell

                cell_type = 'normal_odd'
            else:

            # Even cell
```

```

                     cell_type = 'normal_even'
                 params = ParamArray(
                     cell_intent=cell_type,
                     pwr_width_horiz=self.pwr_width_horiz,
                     pwr_width_vert=self.pwr_width_vert,
                     rc_width=self.rc_width,
                     drain_width=self.drain_width,
                     verticalPitch=self.verticalPitch,
                     W_n=self.W_n,
                     L_n=self.L_n,
                     make_mirrorable='False',
                     extra_vdd_rail_contacts_right=self.extra_vdd_rail_contacts_right,
                     extra_gnd_rail_contacts_right=self.extra_gnd_rail_contacts_right,
                     extra_vdd_rail_contacts_left=self.extra_vdd_rail_contacts_left,
                     extra_gnd_rail_contacts_left=self.extra_gnd_rail_contacts_left,
                     )
                 return [cell_type, params]
```

A.2.4 Control Core

```
#!/usr/bin/python
# -*- coding: utf-8 -*-
from __future__ import with_statement

from BAG_layout import *
import pdb
from operator import itemgetter

# comp_core3.py vs comp_core2.py - comp_core3 moves the level shifter caps
#                                  (pmos in isolated nwells) into the control
#                                  core cell and moves all pmos caps (both
#                                  level shifter and output couplers) to an
#                                  isolated row between the 2 level shifter rows

class comp_core(std_block):

    '''''
    This pycell is the controler for a DCDC switched capacitor regulator.  It consists of an asynchronous comparator
    whose output is preset to a logic high value and drops to logic low after the supply voltage (which is the input)
    drops below a reference voltage.

    Attributes:

    Pycell Parameters:
        L - The default channel length used for devices.
        nov_Wn_tristate - the channel width for the
    '''

    paramNames = dict(
        L='L',
        nov_Wn_tristate='nov_Wn_tristate',
        nov_PNratio='nov_PNratio',
        nov_Wn='nov_Wn',
        nov_Wp='nov_Wp',
        nov_Wn_mid='nov_Wn_mid',
        nov_Wp_mid='nov_Wp_mid',
        reset_inv_Wn='reset_inv_Wn',
        PNratio='PNratio',
        Wn='Wn',
        Wn_clk_driver='Wn_clk_driver',
        toggle_out_width='toggle_out_width',
        out_buffer_width='out_buffer_width',
        Wn_uct_nand='Wn_uct_nand',
        Wn_uct_nand_xor='Wn_uct_nand_xor',
        reset_inv_width='reset_inv_width',
        pullup_width='pullup_width',
        Wn_inv='Wn_inv',
        Wn_uct_buf1='Wn_uct_buf1',
        Wn_uct_buf2='Wn_uct_buf2',
        Wn_uct_inv='Wn_uct_inv',
        ps_Wn_inv='ps_Wn_inv',
        ps_Wn_buf1='ps_Wn_buf1',
        ps_Wn_buf2='ps_Wn_buf2',
        vertical_pitch_center='vertical_pitch_center',
        extra_vertical_pitch='extra_vertical_pitch',
        cell_height='cell_height',
        power_grid_pitch='power_grid_pitch',
        )

    ordered_pins = [  # dict(name='C1_pos_in_phase',justification='LEFT'),
                      # dict(name='VIN', pin_name='VIN2',justification='LEFT'),
                      # dict(name='VOUT', pin_name='VOUT2',justification='LEFT'),
                      # dict(name='GND', pin_name='GND2',justification='LEFT'),
                      # dict(name='C1_pos_out_phase',justification='RIGHT'),
        dict(name='clk1', justification='LEFT'),
        dict(name='clk1_h', justification='LEFT'),
        dict(name='clk2', justification='LEFT'),
        dict(name='clk2_h', justification='LEFT'),
        dict(name='clk2_feedback', justification='LEFT'),
        dict(name='clk_in', justification='CENTER'),
        dict(name='clk_out', justification='CENTER'),
        dict(name='clk_out_buffered', justification='CENTER'),
        dict(name='flip1b_h', justification='CENTER'),
        dict(name='flip1b', justification='CENTER'),
        dict(name='flip2b_h', justification='CENTER'),
        dict(name='flip2b', justification='CENTER'),
        dict(name='clk_xor', justification='CENTER'),
        dict(name='comp_bias_global', justification='CENTER'),
        dict(name='comp_bias', justification='CENTER'),
        dict(name='clk1b_feedback', justification='RIGHT'),
        dict(name='clk1b', justification='RIGHT'),
        dict(name='clk1b_h', justification='RIGHT'),
        dict(name='clk2b', justification='RIGHT'),
        dict(name='clk2b_h', justification='RIGHT'),
        ]

    power_grid = {'pitch': 'power_grid_pitch', 'subelements': [{
        'name': 'VIN',
        'term_type': 'INPUT_OUTPUT',
        'route': 'rail',
        'width': 'vrailWidth',
        'extra_space': 'M4_rail_spacing',
        }, {
        'name': 'VOUT',
        'term_type': 'INPUT_OUTPUT',
        'route': 'rail',
        'width': 'vrailWidth',
        'extra_space': 'M4_rail_spacing',
        }, {
        'name': 'GND',
        'term_type': 'INPUT_OUTPUT',
        'route': 'rail',
        'width': 'vrailWidth',
        'extra_space': 'M4_rail_spacing',
        }]}

    pass_through_pins = [{'name': 'comparator', 'pin': 'int_enable',
                         'term_name': 'comp_int_enable'},
                         {'name': 'comparator', 'pin': 'internal_node',
                         'term_name': 'comp_internal_node'}]

    # types: 'rail', 'route_group' (NORTH to SOUTH list of ordered groups), route, device_row, dev
ice

    pin_info = [  # {'name' : 'VIN',   'pin_name': 'VIN2','term_type' : "INPUT_OUTPUT"  , 'route'
: 'rail', 'width': 'vrailWidth', 'extra_space' : 'M4_rail_spacing' },
                  # {'name' : 'VOUT',  'pin_name': 'VOUT2', 'term_type' :"INPUT_OUTPUT"  , 'route'
 : 'rail', 'width': 'vrailWidth', 'extra_space' : 'M4_rail_spacing' },
                  # {'name' : 'GND',   'pin_name': 'GND2', 'term_type' : "INPUT_OUTPUT"  , 'route'
 : 'rail', 'width': 'vrailWidth', 'extra_space' : 'M4_rail_spacing' },
        {
            'name': 'VIN',
            'term_type': 'INPUT_OUTPUT',
            'route': 'rail',
            'width': 'vrailWidth',
            'extra_space': 'M4_rail_spacing',
            },
        {
            'name': 'VOUT',
            'term_type': 'INPUT_OUTPUT',
            'route': 'rail',
            'width': 'vrailWidth',
            'extra_space': 'M4_rail_spacing',
            },
        {
            'name': 'GND',
            'term_type': 'INPUT_OUTPUT',
            'route': 'rail',
            'width': 'vrailWidth',
            'extra_space': 'M4_rail_spacing',
            },
        {
            'name': 'VREF',
            'term_type': 'INPUT_OUTPUT',
            'route': 'isolated_rail',
            'width': 'vrailWidth',
            'extra_space': 'M4_rail_spacing',
            },
        {
            'name': 'vcomp_pos',
            'term_type': None,
            'route': 'rail',
            'width': 'vrailWidth',
            'extra_space': 'M4_rail_spacing',
            },
        {
            'name': 'clk_out',
            'term_type': 'OUTPUT',
            'route': 'signal',
            'extra_space': 'M4_signal_spacing',
            },
        {
            'name': 'clk_out_buffered',
            'term_type': 'OUTPUT',
            'route': 'signal',
            'extra_space': 'M4_signal_spacing',
            },
        {
            'name': 'clk_xor',
            'term_type': 'OUTPUT',
            'route': 'signal',
            'extra_space': 'M4_signal_spacing',
            },
        {
            'name': 'clk2_h',
            'term_type': 'OUTPUT',
            'route': 'signal',
            'extra_space': 'M4_signal_spacing',
            },
        {
            'name': 'clk1_h',
            'term_type': 'OUTPUT',
            'route': 'signal',
            'extra_space': 'M4_signal_spacing',
            },
        {
            'name': 'clk2',
            'term_type': 'OUTPUT',
            'route': 'signal',
            'extra_space': 'M4_signal_spacing',
            },
        {
            'name': 'clk1',
            'term_type': 'OUTPUT',
            'route': 'signal',
            'extra_space': 'M4_signal_spacing',
            },
        {
            'name': 'clk2b_h',
            'term_type': 'OUTPUT',
            'route': 'signal',
            'extra_space': 'M4_signal_spacing',
            },
        {
            'name': 'clk1b_h',
            'term_type': 'OUTPUT',
            'route': 'signal',
            'extra_space': 'M4_signal_spacing',
            },
        {
            'name': 'clk2b',
            'term_type': 'OUTPUT',
            'route': 'signal',
            'extra_space': 'M4_signal_spacing',
            },
        {
            'name': 'clk1b',
            'term_type': 'OUTPUT',
            'route': 'signal',
            'extra_space': 'M4_signal_spacing',
            },
        {
            'name': 'comp_bias_global',
            'term_type': 'INPUT_OUTPUT',
            'route': 'signal',
            'extra_space': 'M4_signal_spacing',
            },
        {
            'name': 'comp_bias',
            'term_type': 'INPUT_OUTPUT',
            'route': 'signal',
            'extra_space': 'M4_signal_spacing',
            },
        {
            'name': 'clk_in',
            'term_type': 'INPUT',
            'route': 'signal',
            'extra_space': 'M4_signal_spacing',
            },
        {
            'name': 'clk1b_feedback',
            'term_type': 'INPUT',
            'route': 'signal',
            'extra_space': 'M4_signal_spacing',
            },
        {
            'name': 'clk2_feedback',
            'term_type': 'INPUT',
            'route': 'signal',
            'extra_space': 'M4_signal_spacing',
            },
        {
            'name': 'clk_feedback',
            'term_type': 'INPUT_OUTPUT',
            'route': 'signal',
            'extra_space': 'M4_signal_spacing',
            },
        {
            'name': 'toggle_clock',
            'term_type': 'INPUT_OUTPUT',
            'route': 'signal',
            'extra_space': 'M4_signal_spacing',
            },
        {
            'name': 'spi_enable_xor_test',
            'term_type': 'INPUT',
            'route': 'signal',
            'extra_space': 'M4_signal_spacing',
            },
        {
            'name': 'spi_resetb_h',
            'term_type': 'INPUT',
            'route': 'signal',
            'extra_space': 'M4_signal_spacing',
            },
        {
            'name': 'spi_resetb',
            'term_type': 'INPUT',
            'route': 'signal',
            'extra_space': 'M4_signal_spacing',
            },
        {
            'name': 'spi_start',
            'term_type': 'INPUT',
            'route': 'signal',
            'extra_space': 'M4_signal_spacing',
            },
        {
            'name': 'test_clk',
            'term_type': 'INPUT',
            'route': 'signal',
            'extra_space': 'M4_signal_spacing',
            },
        {
            'name': 'spi_toggle_out_enable_b',
            'term_type': 'INPUT',
            'route': 'signal',
            'extra_space': 'M4_signal_spacing',
            },
        {
            'name': 'spi_extra_delay',
            'term_type': 'INPUT',
            'route': 'signal',
            'extra_space': 'M4_signal_spacing',
            },
        {
            'name': 'spi_extra_delay_h',
            'term_type': 'INPUT',
            'route': 'signal',
            'extra_space': 'M4_signal_spacing',
            },
        {
            'name': 'spi_volt_position_conductance_h<4>',
            'term_type': 'INPUT',
            'route': 'signal',
            'extra_space': 'M4_signal_spacing',
            },
        {
            'name': 'spi_volt_position_conductance_h<3>',
            'term_type': 'INPUT',
            'route': 'signal',
            'extra_space': 'M4_signal_spacing',
            },
        {
            'name': 'spi_volt_position_conductance_h<2>',
            'term_type': 'INPUT',
            'route': 'signal',
            'extra_space': 'M4_signal_spacing',
            },
        {
            'name': 'spi_volt_position_conductance_h<1>',
            'term_type': 'INPUT',
            'route': 'signal',
            'extra_space': 'M4_signal_spacing',
            },
        {
            'name': 'spi_volt_position_conductance_h<0>',
            'term_type': 'INPUT',
            'route': 'signal',
            'extra_space': 'M4_signal_spacing',
            },
        {
            'name': 'spi_en_volt_positionb_h',
            'term_type': 'INPUT',
            'route': 'signal',
            'extra_space': 'M4_signal_spacing',
            },
        {
            'name': 'C1_pos_in_phase',
            'term_type': 'INPUT_OUTPUT',
            'route': 'rail',
            'width': 'vrailWidth',
            'extra_space': 'M4_rail_spacing',
            },
        {
            'name': 'C1_pos_out_phase',
            'term_type': 'INPUT_OUTPUT',
            'route': 'rail',
            'width': 'vrailWidth',
            'extra_space': 'M4_rail_spacing',
            },
        {
            'name': 'flip2b_h',
            'term_type': 'INPUT_OUTPUT',
            'route': 'signal',
            'extra_space': 'M4_signal_spacing',
            },
        {
            'name': 'flip2b',
            'term_type': 'INPUT_OUTPUT',
            'route': 'signal',
            'extra_space': 'M4_signal_spacing',
            },
        {
            'name': 'flip1b',
            'term_type': 'INPUT_OUTPUT',
            'route': 'signal',
            'extra_space': 'M4_signal_spacing',
            },
        {
            'name': 'flip1b_h',
            'term_type': 'INPUT_OUTPUT',
            'route': 'signal',
            'extra_space': 'M4_signal_spacing',
            },
        ]

    layout_info = [  # Row 0
                     # This row contains the voltage positioning circuitry which is in the high vo
ltage domain therefore it is in DNW
                     # ('LEFT','RIGHT','CENTER')
                     # #Row 1
                     # #This row contains the high voltage domain portion of the level-
shifter circuits
                     # ('LEFT','RIGHT','CENTER')
                     # #Row 1.5
                     # #This row contains DC blocking caps
                     # ('LEFT','RIGHT','CENTER')
                     # Row 2
                     # Row 3
                     # ('LEFT','RIGHT','CENTER')
                     # {'name' : 'clk_in_inv',
                     # 'lib_cell' : 'BAG_std_cells_oa/Inverter',
                     # 'param_mapping': { 'vertical_pins' : 'vertical_pins',  },
                     # 'pin_mapping' : {   'IN' : 'clk_in',
                     #                     'OUT' : 'clk_inb',
                     #                     'VDD' : 'VOUT',
                     #                     'GND' : 'GND',
                     #                 }
                     # },
                     # #Row 4
                     # 'param_mapping': {  'verticalPitch' : 'vertical_pitch_center', },
                     # 'extra_vertical_pitch_vref' : 'extra_vertical_pitch_vref',
                     # disconnect between vdd rail to left and vdd rail to right
                     # filler should include minimum amount of nwell
                     # this is a dummy pin to force the std_block to connect
                     # the lower level reset net (it isn't connected in the dflop pycell)
        {
            'param_mapping': {'extra_vertical_pitch_bottom': 'volt_position_spacing'
                              , 'deep_nwell': 'true_value',
                              'verticalPitch': 'vertical_pitch_volt_position'
                              },
            'mirror_x': 'False',
            'justification': 'LEFT',
            'subelements': [{
                'name': 'volt_position',
                'lib_cell': 'BAG_std_cells_oa/volt_position',
                'mirror_y': 'False',
                'mirror_x': 'False',
                'param_mapping': {'share_pwell_left': 'false_value',
                                  'share_pwell_right': 'false_value'},
                'pin_mapping': {
                    'en_volt_positionb_h_left': 'spi_en_volt_positionb_h'
                        ,
                    'in_phase_en_h': 'clk1_h',
                    'volt_position_conductance_h<4>': 'spi_volt_position_conductance_h<4>',
                    'volt_position_conductance_h<3>': 'spi_volt_position_conductance_h<3>',
                    'volt_position_conductance_h<2>': 'spi_volt_position_conductance_h<2>',
                    'volt_position_conductance_h<1>': 'spi_volt_position_conductance_h<1>',
                    'volt_position_conductance_h<0>': 'spi_volt_position_conductance_h<0>',
                    'C1_pos_in_phase': 'C1_pos_in_phase',
                    'C1_pos_out_phase': 'C1_pos_out_phase',
                    'en_volt_positionb_h_right': 'spi_en_volt_positionb_h'
                        ,
                    'out_phase_en_h': 'clk2b_h',
                    'vcomp_pos': 'vcomp_pos',
                    'VDD': 'VIN',
                    'GND': 'VOUT',
                    },
                }],
            },
        {
            'param_mapping': {'deep_nwell': 'true_value',
                              'verticalPitch': 'vertical_pitch_nov'},
            'mirror_x': 'True',
            'justification': 'LEFT',
            'subelements': [{
                'name': 'nov_1',
                'lib_cell': 'BAG_std_cells_oa/nov_level_shift_2',
                'mirror_y': 'True',
                'simple_filler': 'True',
                'param_mapping': {
                    'L': 'L',
                    'Wn_tristate': 'nov_Wn_tristate',
                    'PNratio': 'nov_PNratio',
                    'share_pwell_right': 'false_value',
                    'Wn': 'nov_Wn',
                    'Wp': 'nov_Wp',
                    'Wn_mid': 'nov_Wn_mid',
                    'Wp_mid': 'nov_Wp_mid',
                    'extra_nwell_space': 'extra_nwell_space',
                    },
                'pin_mapping': {
                    'outb_slow_fall': 'clk2_h',
                    'outb_slow_rise': 'clk1_h',
                    'in': 'flip1b_h',
                    'extra_delay': 'spi_extra_delay_h',
                    'reset': 'reset_h',
                    'resetb': 'resetb_h',
                    'setb': 'VOUT',
                    'vdd': 'VIN',
                    'vss': 'VOUT',
                    'buffered_ls_signal': 'buffered_ls_signal_1',
                    'feedback_2': 'feedback_2_1',
                    'fast_feedback_pmos': 'fast_feedback_pmos_1',
                    'fast_feedback_nmos': 'fast_feedback_nmos_1',
                    'mux_input_a2': 'mux_input_a2_1',
                    },
                }, {
                'name': 'reset_h_pre_inv',
                'lib_cell': 'BAG_std_cells_oa/Inverter',
                'mirror_y': 'False',
                'param_mapping': {'Wn': 'reset_inv_Wn',
                                  'PNratio': 'PNratio',
                                  'vertical_pins': 'vertical_pins'},
                'pin_mapping': {
                    'IN': 'spi_resetb_h',
                    'OUT': 'reseth_int1',
                    'VDD': 'VIN',
                    'GND': 'VOUT',
                    },
                }, {
                'name': 'resetb_h_inv',
                'lib_cell': 'BAG_std_cells_oa/Inverter',
                'mirror_y': 'False',
                'param_mapping': {'Wn': 'reset_inv_Wn',
                                  'PNratio': 'PNratio',
                                  'vertical_pins': 'vertical_pins'},
                'pin_mapping': {
                    'IN': 'reseth_int1',
                    'OUT': 'resetb_h',
                    'VDD': 'VIN',
                    'GND': 'VOUT',
                    },
                }, {
                'name': 'reset_h_inv',
                'lib_cell': 'BAG_std_cells_oa/Inverter',
                'mirror_y': 'False',
                'param_mapping': {'Wn': 'reset_inv_Wn',
                                  'PNratio': 'PNratio',
                                  'vertical_pins': 'vertical_pins'},
                'pin_mapping': {
                    'IN': 'resetb_h',
                    'OUT': 'reset_h',
                    'VDD': 'VIN',
                    'GND': 'VOUT',
                    },
                }, {
                'name': 'nov_4',
                'lib_cell': 'BAG_std_cells_oa/nov_level_shift_2',
                'mirror_y': 'False',
                'param_mapping': {
                    'L': 'L',
                    'Wn_tristate': 'nov_Wn_tristate',
                    'PNratio': 'nov_PNratio',
                    'share_pwell_right': 'false_value',
                    'Wn': 'nov_Wn',
                    'Wp': 'nov_Wp',
                    'Wn_mid': 'nov_Wn_mid',
                    'Wp_mid': 'nov_Wp_mid',
                    'extra_nwell_space': 'extra_nwell_space',
                    },
                'pin_mapping': {
                    'outb_slow_fall': 'clk1b_h',
                    'outb_slow_rise': 'clk2b_h',
                    'in': 'flip2b_h',
                    'extra_delay': 'spi_extra_delay_h',
                    'reset': 'reset_h',
                    'resetb': 'resetb_h',
                    'setb': 'VIN',
                    'vdd': 'VIN',
                    'vss': 'VOUT',
                    'buffered_ls_signal': 'buffered_ls_signal_4',
                    'feedback_2': 'feedback_2_4',
                    'fast_feedback_pmos': 'fast_feedback_pmos_4',
                    'fast_feedback_nmos': 'fast_feedback_nmos_4',
                    'mux_input_a2': 'mux_input_a2_4',
                    },
                }],
            },
        {'justification': 'LEFT',
         'param_mapping': {'extra_vertical_pitch_top': 'extra_vertical_pitch_dnw'
         , 'verticalPitch': 'vertical_pitch_caps'}, 'subelements': [
            {
                'name': 'M_clk1_coupling',
                'lib_cell': 'BAG_std_cells_oa/pmos_cap_m2',
                'simple_filler': 'True',
                'param_mapping': {
                    'vdd_rail_pin_in_metal': 'number_one',
                    'deep_nwell': 'false_value',
                    'share_nwell_right': 'share_nwell_pmos_cap',
                    'share_nwell_left': 'share_nwell_pmos_cap',
                    'extend_enclosing_layers_to_pr': 'extend_enclosing_layers_pmos_cap'
                    ,
                    'nmos_ratio': 'nmos_ratio_pmos_cap',
                    },
                'pin_mapping': {
                    'PLUS': 'clk1_h',
                    'PLUS_rail': None,
                    'MINUS': 'clk1',
                    'GND': 'GND',
                    },
                },
            {
                'name': 'M_clk2_coupling',
                'lib_cell': 'BAG_std_cells_oa/pmos_cap_m2',
            'simple_filler': 'True',
            'param_mapping': {
                'vdd_rail_pin_in_metal': 'number_one',
                'share_nwell_right': 'share_nwell_pmos_cap',
                'share_nwell_left': 'share_nwell_pmos_cap',
                'extend_enclosing_layers_to_pr': 'extend_enclosing_layers_pmos_cap'
                    ,
                'nmos_ratio': 'nmos_ratio_pmos_cap',
                },
            'pin_mapping': {
                'PLUS': 'clk2_h',
                'PLUS_rail': None,
                'MINUS': 'clk2',
                'GND': 'GND',
                },
            },
        {
            'name': 'cap_ls_1',
            'lib_cell': 'BAG_std_cells_oa/pmos_cap_m2',
            'simple_filler': 'True',
            'param_mapping': {
                'vdd_rail_pin_in_metal': 'number_one',
                'share_nwell_right': 'share_nwell_pmos_cap',
                'share_nwell_left': 'share_nwell_pmos_cap',
                'extend_enclosing_layers_to_pr': 'extend_enclosing_layers_pmos_cap'
                    ,
                'nmos_ratio': 'nmos_ratio_pmos_cap',
                },
            'pin_mapping': {
                'PLUS': 'flip2b_h',
                'PLUS_rail': None,
                'MINUS': 'flip2b',
                'GND': 'GND',
                },
            },
        {
            'name': 'cap_ls_2',
            'lib_cell': 'BAG_std_cells_oa/pmos_cap_m2',
            'mirror_y': 'True',
            'simple_filler': 'True',
            'param_mapping': {
                'vdd_rail_pin_in_metal': 'number_one',
                'share_nwell_right': 'share_nwell_pmos_cap',
                'share_nwell_left': 'share_nwell_pmos_cap',
                'extend_enclosing_layers_to_pr': 'extend_enclosing_layers_pmos_cap'
                    ,
                'nmos_ratio': 'nmos_ratio_pmos_cap',
                },
            'pin_mapping': {
                'PLUS': 'flip1b_h',
                'PLUS_rail': None,
                'MINUS': 'flip1b',
                'GND': 'GND',
                },
            },
        {
            'name': 'M_clk1b_coupling',
            'lib_cell': 'BAG_std_cells_oa/pmos_cap_m2',
            'mirror_y': 'True',
            'simple_filler': 'True',
            'param_mapping': {
                'vdd_rail_pin_in_metal': 'number_one',
                'deep_nwell': 'false_value',
                'share_nwell_right': 'share_nwell_pmos_cap',
                'share_nwell_left': 'share_nwell_pmos_cap',
                'extend_enclosing_layers_to_pr': 'extend_enclosing_layers_pmos_cap'
                    ,
                'nmos_ratio': 'nmos_ratio_pmos_cap',
                },
            'pin_mapping': {
                'PLUS': 'clk1b_h',
                'PLUS_rail': None,
                'MINUS': 'clk1b',
                'GND': 'GND',
                },
            },
        {
            'name': 'M_clk2b_coupling',
            'lib_cell': 'BAG_std_cells_oa/pmos_cap_m2',
            'mirror_y': 'True',
            'simple_filler': 'True',
            'param_mapping': {
                'vdd_rail_pin_in_metal': 'number_one',
                'share_nwell_right': 'share_nwell_pmos_cap',
                'share_nwell_left': 'share_nwell_pmos_cap',
                'extend_enclosing_layers_to_pr': 'extend_enclosing_layers_pmos_cap'
                    ,
                'nmos_ratio': 'nmos_ratio_pmos_cap',
                },
            'pin_mapping': {
                'PLUS': 'clk2b_h',
                'PLUS_rail': None,
                'MINUS': 'clk2b',
                'GND': 'GND',
                },
            },
        ]},
    {'param_mapping': {'verticalPitch': 'vertical_pitch_nov'},
     'mirror_x': 'True', 'subelements': [{
        'name': 'nov_2',
        'lib_cell': 'BAG_std_cells_oa/nov_level_shift_2',
        'mirror_y': 'True',
        'simple_filler': 'True',
        'param_mapping': {
            'L': 'L',
            'Wn_tristate': 'nov_Wn_tristate',
            'PNratio': 'nov_PNratio',
            'Wn': 'nov_Wn',
            'Wp': 'nov_Wp',
            'Wn_mid': 'nov_Wn_mid',
            'Wp_mid': 'nov_Wp_mid',
            },
        'pin_mapping': {
            'outb_slow_fall': 'clk2',
            'outb_slow_rise': 'clk1',
            'in': 'flip1b',
            'extra_delay': 'spi_extra_delay',
            'reset': 'reset',
            'resetb': 'resetb',
            'setb': 'GND',
            'vdd': 'VOUT',
            'vss': 'GND',
            'buffered_ls_signal': 'buffered_ls_signal_2',
            'feedback_2': 'feedback_2_2',
            'fast_feedback_pmos': 'fast_feedback_pmos_2',
            'fast_feedback_nmos': 'fast_feedback_nmos_2',
            'mux_input_a2': 'mux_input_a2_2',
            },
        }, {
        'name': 'reset_pre_inv',
        'lib_cell': 'BAG_std_cells_oa/Inverter',
        'mirror_y': 'True',
        'param_mapping': {'Wn': 'reset_inv_Wn', 'PNratio': 'PNratio'
                          , 'vertical_pins': 'vertical_pins'},
        'pin_mapping': {
            'IN': 'spi_resetb',
            'OUT': 'reset_int1',
            'VDD': 'VOUT',
            'GND': 'GND',
            },
        }, {
        'name': 'resetb_inv',
        'lib_cell': 'BAG_std_cells_oa/Inverter',
        'mirror_y': 'True',
        'param_mapping': {'Wn': 'reset_inv_Wn', 'PNratio': 'PNratio'
                          , 'vertical_pins': 'vertical_pins'},
        'pin_mapping': {
            'IN': 'reset_int1',
            'OUT': 'resetb',
            'VDD': 'VOUT',
            'GND': 'GND',
            },
        }, {
        'name': 'reset_inv',
        'lib_cell': 'BAG_std_cells_oa/Inverter',
        'mirror_y': 'False',
        'param_mapping': {'Wn': 'reset_inv_Wn', 'PNratio': 'PNratio'
                          , 'vertical_pins': 'vertical_pins'},
        'pin_mapping': {
            'IN': 'resetb',
            'OUT': 'reset',
            'VDD': 'VOUT',
            'GND': 'GND',
            },
        }, {
        'name': 'nov_3',
        'lib_cell': 'BAG_std_cells_oa/nov_level_shift_2',
        'mirror_y': 'False',
        'param_mapping': {
            'L': 'L',
            'Wn_tristate': 'nov_Wn_tristate',
            'PNratio': 'nov_PNratio',
            'Wn': 'nov_Wn',
            'Wp': 'nov_Wp',
            'Wn_mid': 'nov_Wn_mid',
            'Wp_mid': 'nov_Wp_mid',
            },
        'pin_mapping': {
            'outb_slow_fall': 'clk1b',
            'outb_slow_rise': 'clk2b',
            'in': 'flip2b',
            'extra_delay': 'spi_extra_delay',
            'reset': 'reset',
            'resetb': 'resetb',
            'setb': 'VOUT',
            'vdd': 'VOUT',
            'vss': 'GND',
            'buffered_ls_signal': 'buffered_ls_signal_3',
            'feedback_2': 'feedback_2_3',
            'fast_feedback_pmos': 'fast_feedback_pmos_3',
            'fast_feedback_nmos': 'fast_feedback_nmos_3',
            'mux_input_a2': 'mux_input_a2_3',
            },
        }]},
    {
        'param_mapping': {},
        'mirror_x': 'False',
        'justification': 'LEFT',
        'subelements': [
            {
                'name': 'uct_nand1',
                'lib_cell': 'BAG_std_cells_oa/Nand',
                'param_mapping': {
                    'L': 'L',
                    'Wn': 'Wn_uct_nand',
                    'PNratio': 'pnrat_nand',
                    'vertical_pins': 'vertical_pins',
                    },
                'pin_mapping': {
                    'IN1': 'spi_enable_xor_test',
                    'IN2': 'clk_in',
                    'OUT': 'uct_clk_in_b',
                    'VDD': 'VOUT',
                    'GND': 'GND',
                    },
                },
            {
                'name': 'uct_inv1',
                'lib_cell': 'BAG_std_cells_oa/Inverter',
                'param_mapping': {
                    'L': 'L',
                    'Wn': 'Wn_uct_inv',
                    'PNratio': 'pnrat_inv',
                    'vertical_pins': 'vertical_pins',
                    },
                'pin_mapping': {
                    'IN': 'uct_clk_in_b',
                    'OUT': 'uct_clk_in',
                    'VDD': 'VOUT',
                    'GND': 'GND',
                    },
                },
            {
                'name': 'uct_nand2',
                'lib_cell': 'BAG_std_cells_oa/Nand',
                'param_mapping': {'Wn': 'Wn_uct_nand',
                        'PNratio': 'pnrat_nand',
                        'vertical_pins': 'vertical_pins'},
                'pin_mapping': {
                    'IN1': 'spi_enable_xor_test',
                    'IN2': 'clk_feedback',
                    'OUT': 'uct_clk_out_b',
                    'VDD': 'VOUT',
                    'GND': 'GND',
                    },
                },
            {
                'name': 'uct_inv2',
                'lib_cell': 'BAG_std_cells_oa/Inverter',
                'param_mapping': {'Wn': 'Wn_uct_inv',
                        'PNratio': 'pnrat_inv',
                        'vertical_pins': 'vertical_pins'},
                'pin_mapping': {
                    'IN': 'uct_clk_out_b',
                    'OUT': 'uct_clk_out',
                    'VDD': 'VOUT',
                    'GND': 'GND',
                    },
                },
            {
                'name': 'uct_dflop_1',
                'lib_cell': 'BAG_std_cells_oa/dflop',
                'param_mapping': {},
                'pin_mapping': {
                    'clk': 'uct_clk_in',
                    'clk_b': 'uct_dflop_1_clk_b_connect',
                    'reset': 'uct_dflop_1_reset_connect',
                    'int1': 'uct_dflop_1_int1_connect',
                    'resetb': 'resetb',
                    'Q': 'uct_Q1',
                    'D': 'test_clk',
                    'vdd': 'VOUT',
                    'vss': 'GND',
                    },
                },
            {
                'name': 'uct_dflop_2',
                'lib_cell': 'BAG_std_cells_oa/dflop',
                'param_mapping': {},
                'pin_mapping': {
                    'clk': 'uct_clk_out',
                    'clk_b': 'uct_dflop_2_clk_b_connect',
                    'reset': 'uct_dflop_2_reset_connect',
                    'int1': 'uct_dflop_2_int1_connect',
                    'resetb': 'resetb',
                    'Q': 'uct_Q2',
                    'D': 'test_clk',
                    'vdd': 'VOUT',
                    'vss': 'GND',
                    },
                },
            {
                'name': 'start_inv',
                'lib_cell': 'BAG_std_cells_oa/Inverter',
                'param_mapping': {'Wn': 'Wn', 'PNratio': 'pnrat_inv'
                        , 'vertical_pins': 'vertical_pins'},
                'pin_mapping': {
                    'IN': 'spi_start',
                    'OUT': 'startb_int1',
                    'VDD': 'VOUT',
                    'GND': 'GND',
                    },
                },
            {
                'name': 'start_inv2',
                'lib_cell': 'BAG_std_cells_oa/Inverter',
                'param_mapping': {'Wn': 'Wn', 'PNratio': 'pnrat_inv'
                        , 'vertical_pins': 'vertical_pins'},
                'pin_mapping': {
                    'IN': 'startb_int1',
                    'OUT': 'startb_int2',
                    'VDD': 'VOUT',
                    'GND': 'GND',
                    },
                },
            {
                'name': 'start_inv3',
                'lib_cell': 'BAG_std_cells_oa/Inverter',
                'param_mapping': {'Wn': 'Wn', 'PNratio': 'pnrat_inv'
                        , 'vertical_pins': 'vertical_pins'},
                'pin_mapping': {
                    'IN': 'startb_int2',
                    'OUT': 'startb',
                    'VDD': 'VOUT',
                    'GND': 'GND',
                    },
                },
            {
                'name': 'pull_up_start',
                'lib_cell': 'BAG_std_cells_oa/falling_edge_detect_pull_up'
                    ,
                'param_mapping': {'pullup_width': 'pullup_width',
                        'Wn_inv': 'Wn_inv'},
                'pin_mapping': {
                    'in': 'startb',
                    'out': 'clk_feedback',
                    'VDD': 'VOUT',
                    'GND': 'GND',
                    },
                },
            {
                'name': 'inv_pre2',
                'lib_cell': 'BAG_std_cells_oa/Inverter',
                'param_mapping': {
                    'L': 'L',
                    'Wn': 'Wn_inv',
                    'PNratio': 'pnrat_inv',
                    'vertical_pins': 'vertical_pins',
                    },
                'pin_mapping': {
                    'IN': 'clk1b_feedback',
                    'OUT': 'clk1b_feedback_b',
                    'VDD': 'VOUT',
                    'GND': 'GND',
                    },
                },
            {
                'name': 'nand_pre2',
                'lib_cell': 'BAG_std_cells_oa/Nand',
                'param_mapping': {
                    'L': 'L',
                    'Wn': 'Wn_inv',
                    'PNratio': 'pnrat_nand',
                    'vertical_pins': 'vertical_pins',
                    },
                'pin_mapping': {
                    'IN1': 'clk2_feedback',
                    'IN2': 'clk1b_feedback_b',
                    'OUT': 'clk1b_feedback_gated',
                    'VDD': 'VOUT',
                    'GND': 'GND',
                    },
                },
            {
                'name': 'pull_up_long_1',
                'lib_cell': 'BAG_std_cells_oa/falling_edge_detect_pull_up'
                    ,
                'param_mapping': {'pullup_width': 'pullup_width',
                        'Wn_inv': 'Wn_inv'},
                'pin_mapping': {
                    'in': 'clk1b_feedback_gated',
                    'out': 'clk_feedback',
                    'VDD': 'VOUT',
                    'GND': 'GND',
                    },
                },
            {
                'name': 'clk_driver',
                'lib_cell': 'BAG_std_cells_oa/clk_driver',
                'param_mapping': {'Wn': 'Wn_clk_driver',
                        'extra_vdd_rail_contacts_right': 'number_five'
                        ,
                        'extra_gnd_rail_contacts_right': 'number_five'
                        },
                'pin_mapping': {
                    'reset': 'reset',
                    'resetb': 'resetb',
                    'clk_out': 'clk_feedback',
                    'clk_next': 'toggle_clockb_delayed',
                    'clk_outb': 'clk_outb',
                    'startb': 'startb',
                    'VDD': 'VOUT',
                    'GND': 'GND',
                    },
                },
            {
                'name': 'pull_up_long_2',
                'lib_cell': 'BAG_std_cells_oa/falling_edge_detect_pull_up'
                    ,
                'param_mapping': {'pullup_width': 'pullup_width',
                        'Wn_inv': 'Wn_inv'},
                'pin_mapping': {
                    'in': 'clk2_feedback_gated',
                    'out': 'clk_feedback',
                    'VDD': 'VOUT',
                    'GND': 'GND',
                    },
                },
            {
                        'name': 'nand_pre1',
                        'lib_cell': 'BAG_std_cells_oa/Nand',
                        'param_mapping': {
                            'L': 'L',
                            'Wn': 'Wn_inv',
                            'PNratio': 'pnrat_nand',
                            'vertical_pins': 'vertical_pins',
                            },
                            'pin_mapping': {
                                'IN1': 'clk1b_feedback',
                                'IN2': 'clk2_feedback_b',
                                'OUT': 'clk2_feedback_gated',
                                'VDD': 'VOUT',
                                'GND': 'GND',
                                },
                            },
                        {
                            'name': 'inv_pre1',
                            'lib_cell': 'BAG_std_cells_oa/Inverter',
                            'param_mapping': {
                                'L': 'L',
                                'Wn': 'Wn_inv',
                                'PNratio': 'pnrat_inv',
                                'vertical_pins': 'vertical_pins',
                                },
                            'pin_mapping': {
                                'IN': 'clk2_feedback',
                                'OUT': 'clk2_feedback_b',
                                'VDD': 'VOUT',
                                'GND': 'GND',
                                },
                            },
                        {
                            'name': 'toggle_out_nor',
                            'lib_cell': 'BAG_std_cells_oa/Nor',
                            'param_mapping': {'Wn': 'toggle_out_width',
                                    'PNratio': 'PNratio',
                                    'vertical_pins': 'vertical_pins'},
                            'pin_mapping': {
                                'IN1': 'spi_toggle_out_enable_b',
                                'IN2': 'toggle_clockb_delayed',
                                'OUT': 'clk_drv_nor_int',
                                'VDD': 'VOUT',
                                'GND': 'GND',
                                },
                            },
                        {
                            'name': 'out_buffer',
                            'lib_cell': 'BAG_std_cells_oa/Nor',
                            'param_mapping': {'Wn': 'out_buffer_width',
                                    'PNratio': 'PNratio',
                                    'vertical_pins': 'vertical_pins'},
                            'pin_mapping': {
                                'IN1': 'clk_drv_nor_int',
                                'IN2': 'clk_feedback',
                                'OUT': 'clk_outb',
                                'VDD': 'VOUT',
                                'GND': 'GND',
                                },
                            },
                        {
                            'name': 'output_inverter_3',
                            'lib_cell': 'BAG_std_cells_oa/Inverter',
                            'param_mapping': {'vertical_pins': 'vertical_pins'
                                    },
                            'pin_mapping': {
                                'IN': 'clk_outb',
                                'OUT': 'clk_out_int1',
                                'VDD': 'VOUT',
                                'GND': 'GND',
                            },
                        },
                        {
                            'name': 'output_inverter_2',
                            'lib_cell': 'BAG_std_cells_oa/Inverter',
                            'param_mapping': {'vertical_pins': 'vertical_pins'
                                    },
                            'pin_mapping': {
                                'IN': 'clk_out_int1',
                                'OUT': 'clk_out_int2',
                                'VDD': 'VOUT',
                                'GND': 'GND',
                            },
                        },
                        {
                            'name': 'output_inverter',
                            'lib_cell': 'BAG_std_cells_oa/Inverter',
                            'param_mapping': {'vertical_pins': 'vertical_pins'
                                    },
                            'pin_mapping': {
                                'IN': 'clk_out_int2',
                                'OUT': 'clk_out_buffered',
                                'VDD': 'VOUT',
                                'GND': 'GND',
                            },
                        },
                        {
                            'name': 'nand_xor1',
                            'lib_cell': 'BAG_std_cells_oa/Nand',
                            'param_mapping': {'Wn': 'Wn_uct_nand_xor',
                                    'PNratio': 'pnrat_nand',
                                    'vertical_pins': 'vertical_pins'},
                            'pin_mapping': {
                                'IN1': 'uct_Q2',
                                'IN2': 'uct_Q1',
                                'OUT': 'xor_int1',
                                'VDD': 'VOUT',
                                'GND': 'GND',
                            },
                        },
                        {
                            'name': 'nand_xor2',
                            'lib_cell': 'BAG_std_cells_oa/Nand',
                            'param_mapping': {'Wn': 'Wn_uct_nand_xor',
                                    'PNratio': 'pnrat_nand',
                                    'vertical_pins': 'vertical_pins'},
                            'pin_mapping': {
                                'IN1': 'xor_int1',
                                'IN2': 'uct_Q1',
                                'OUT': 'xor_int2',
                                'VDD': 'VOUT',
                                'GND': 'GND',
                            },
                        },
                        {
                            'name': 'nand_xor3',
                            'lib_cell': 'BAG_std_cells_oa/Nand',
                            'param_mapping': {'Wn': 'Wn_uct_nand_xor',
                                    'PNratio': 'pnrat_nand',
                                    'vertical_pins': 'vertical_pins'},
                            'pin_mapping': {
                                'IN1': 'uct_Q2',
                                'IN2': 'xor_int1',
                                'OUT': 'xor_int3',
                                'VDD': 'VOUT',
                                'GND': 'GND',
                                },
                            },
                        {
                            'name': 'nand_xor4',
                            'lib_cell': 'BAG_std_cells_oa/Nand',
                            'param_mapping': {'Wn': 'Wn_uct_nand_xor',
                                    'PNratio': 'pnrat_nand',
                                    'vertical_pins': 'vertical_pins'},
                            'pin_mapping': {
                                'IN1': 'xor_int3',
                                'IN2': 'xor_int2',
                                'OUT': 'xor_out',
                                'VDD': 'VOUT',
                                'GND': 'GND',
                                },
                            },
                        {
                            'name': 'uct_dflop_3',
                            'lib_cell': 'BAG_std_cells_oa/dflop',
                            'param_mapping': {},
                            'pin_mapping': {
                                'clk': 'uct_clk_in',
                                'clk_b': 'uct_dflop_3_clk_b_connect',
                                'reset': 'uct_dflop_3_reset_connect',
                                'int1': 'uct_dflop_3_int1_connect',
                                'resetb': 'resetb',
                                'Q': 'clk_xor_pre',
                                'D': 'xor_out',
                                'vdd': 'VOUT',
                                'vss': 'GND',
                                },
                            },
                        {
                            'name': 'uct_out_buf1',
                            'lib_cell': 'BAG_std_cells_oa/Inverter',
                            'param_mapping': {'Wn': 'Wn_uct_buf1',
                                    'PNratio': 'pnrat_inv',
                                    'vertical_pins': 'vertical_pins'},
                            'pin_mapping': {
                                'IN': 'clk_xor_pre',
                                'OUT': 'clk_xor_b',
                                'VDD': 'VOUT',
                                'GND': 'GND',
                                },
                            },
                        {
                            'name': 'uct_out_buf2',
                            'lib_cell': 'BAG_std_cells_oa/Inverter',
                            'param_mapping': {'Wn': 'Wn_uct_buf2',
                                    'PNratio': 'pnrat_inv',
                                    'vertical_pins': 'vertical_pins'},
                            'pin_mapping': {
                                'IN': 'clk_xor_b',
                                'OUT': 'clk_xor',
                                'VDD': 'VOUT',
                                'GND': 'GND',
                                },
                            },
                        ],
                    },
                {'mirror_x': 'True', 'subelements': [{
                    'name': 'comparator',
                    'lib_cell': 'BAG_std_cells_oa/async_comp',
                    'param_mapping': {'nmos_ratio': 'comparator_nmos_ratio'},
                    'pin_mapping': {
                        'bias': 'comp_bias',
                        'bias2': 'comp_bias_global',
                        'enable': 'clk_feedback',
                        'out': 'toggle_clock',
                        'VREF': 'VREF',
                        'VDD': 'vcomp_pos',
                        'GND': 'GND',
                        'int_enable': 'comp_int_enable',
                        'internal_node': 'comp_internal_node',
                        },
                    }, {
                    'name': 'toggle_clock_delay',
                    'type': 'std_cell',
                    'lib_cell': 'BAG_std_cells_oa/inverting_delay',
                    'simple_filler': 'True',
                    'param_mapping': {'vertical_pins': 'vertical_pins',
                                      'share_nwell_left': 'false_value'},
                    'pin_mapping': {
                        'IN': 'toggle_clock',
                        'OUT': 'toggle_clockb_delayed',
                        'VDD': 'VOUT',
                        'GND': 'GND',
                        },
                    }, {
                    'name': 'dflop',
                    'lib_cell': 'BAG_std_cells_oa/dflop_cp',
                    'param_mapping': {'share_nwell_left': 'false_value'},
                    'simple_filler': 'True',
                    'pin_mapping': {
                        'clk': 'dflop_clk_connect',
                        'clkb': 'toggle_clock',
                        'clkb_buf': 'dflop_clkb_buf_connect',
                        'reset': 'dflop_reset_connect',
                        'int1': 'dflop_int1_connect',
                        'resetb': 'resetb',
                        'Q': 'clk_out',
                        'D': 'D',
                        'vdd': 'VOUT',
                        'vss': 'GND',
                        },
                    }, {
                    'name': 'feedback_inv',
                    'lib_cell': 'BAG_std_cells_oa/Inverter',
                    'param_mapping': {'vertical_pins': 'vertical_pins'},
                    'pin_mapping': {
                        'IN': 'clk_out',
                        'OUT': 'D',
                        'VDD': 'VOUT',
                        'GND': 'GND',
                        },
                    }, {
                    'name': 'phase_splitter',
                    'lib_cell': 'BAG_std_cells_oa/phase_splitter_21',
                    'param_mapping': {'Wn_inv': 'ps_Wn_inv',
                                      'Wn_buf1': 'ps_Wn_buf1',
                                      'Wn_buf2': 'ps_Wn_buf2'},
                    'pin_mapping': {
                        'IN': 'clk_in',
                        'OUT': 'flip2b',
                        'OUTB': 'flip1b',
                        'VDD': 'VOUT',
                        'VSS': 'GND',
                        },
                    }]},
                ]

            default = dict(
                Wn=1.0,
                PNratio=2.0,
                nov_PNratio=2.0,
                railWidth=0.6,
                rail_pins_in_metal=3,
                power_grid_pitch=2.5,
                param_file='/tools/designs/crossley/BAG_2012/BAG_tsmc65/pycell_work/comp_core_file
.yaml',
                cell_height=50.0,
                L=0.06,
                )

             # ###################################################################

            @classmethod
            def defineParamSpecs(cls, specs):
                """Define the PyCell parameters.  The order of invocation of
                specs() becomes the order on the form.

                Arguments:
                specs - (ParamSpecArray)  PyCell parameters
                    """

                super(comp_core, cls).defineParamSpecs(specs)
                mySpecs = ParamSpecArray()

                minmetal1 = float(specs.tech.getPhysicalRule('minWidth',
                                  cls.layer['metal1']))


                Wn = specs.tech.getMosfetParams('nmos', cls.oxide, 'minWidth')
                Wn = cls.default.get('Wn', Wn)
                Wn = cls.grid.snap(Wn)
                stepConstraint = StepConstraint(cls.maskgrid, start=Wn,
                                                resolution=cls.resolution,
                                                action=FailAction.REJECT)
                mySpecs('Wn', Wn, constraint=stepConstraint)

                params_using_Wn_as_default = [
                    'nov_Wn',
                    'nov_Wn_tristate',
                    'nov_Wp',
                    'nov_Wn_mid',
                    'nov_Wp_mid',
                    'Wn_clk_driver',
                    'toggle_out_width',
                    'out_buffer_width',
                    'Wn_uct_nand',
                    'Wn_uct_nand_xor',
                    'reset_inv_Wn',
                    'reset_inv_width',
                    'pullup_width',
                    'Wn_inv',
                    'Wn_uct_buf1',
                    'Wn_uct_buf2',
                    'Wn_uct_inv',
                    ]

                for the_p in params_using_Wn_as_default:
                    mySpecs(the_p, cls.default.get(the_p, Wn),
                            constraint=stepConstraint)

                mySpecs('ps_Wn_inv', cls.default.get('ps_Wn_inv', 4 * Wn),
                        constraint=stepConstraint)
                mySpecs('ps_Wn_buf1', cls.default.get('ps_Wn_buf1', 4 * Wn),
                        constraint=stepConstraint)
                mySpecs('ps_Wn_buf2', cls.default.get('ps_Wn_buf2', 4 * Wn),
                        constraint=stepConstraint)

                L = specs.tech.getMosfetParams('nmos', cls.oxide, 'minLength')
                L = cls.grid.snap(L)
                stepConstraint = StepConstraint(cls.maskgrid, start=L,
                        resolution=cls.resolution, action=FailAction.REJECT)
                mySpecs('L', cls.default.get('L', L), constraint=stepConstraint)

                PNratio = cls.default.get('PNratio', 1.0)
                mySpecs('PNratio', PNratio, constraint=StepConstraint(0.01,
                        start=0.2, action=FailAction.REJECT))

                nov_PNratio = cls.default.get('nov_PNratio', 1.0)
                mySpecs('nov_PNratio', nov_PNratio,
                        constraint=StepConstraint(0.01, start=0.2,
                        action=FailAction.REJECT))

                vertical_pitch_center = cls.default.get('vertical_pitch_center'
                        , 60.0 * minmetal1)
                mySpecs('vertical_pitch_center', vertical_pitch_center,
                        constraint=StepConstraint(cls.maskgrid,
                        start=minmetal1, resolution=cls.resolution,
                        action=FailAction.REJECT))

                cell_height = cls.default.get('cell_height', 62.25)
                mySpecs('cell_height', cell_height,
                        constraint=StepConstraint(cls.maskgrid,
                        start=minmetal1, resolution=cls.resolution,
                        action=FailAction.REJECT))

                power_grid_pitch = cls.default.get('power_grid_pitch', 2.5)
                mySpecs('power_grid_pitch', power_grid_pitch,
                        constraint=StepConstraint(cls.maskgrid, start=minmetal1
                        * 2, resolution=cls.resolution,
                        action=FailAction.REJECT))

                mySpecs('extra_vertical_pitch', 0.0,
                        constraint=RangeConstraint(0.0, None,
                        action=FailAction.REJECT))

                # Parameter renaming

                renameParams(mySpecs, specs, cls.paramNames)

            # ###################################################################

            def final_steps(self):
                '''''
                '''

                for (row_index, row) in enumerate(self.layout_info):
                    print (row_index, 'row width is:', row['group'
                           ].getBBox(ShapeFilter([self.layer['pr'
                           ]])).getWidth())
                for (row_index, row) in enumerate(self.layout_info):
                    print (row_index, 'row height is:', row['group'
                           ].getBBox(ShapeFilter([self.layer['pr'
                           ]])).getHeight())

            def define_derived_parameters(self):
                '''''
                Any derived parameters that are referenced in self.layout_info
                should be defined here

                For example, in the pmos device_group below the param_mapping of
                'W' is 'Wp'.  If Wp is not a parameter defined a the pycell level,
                 then self.Wp must be defined in this function.
                                  'subelements' : [  {  'name':'M3M4',

                                                        'param_mapping': {'W':'Wp' },
                                                        'G' : ['IN1','IN2'],
                                                        'D' : ['OUT'],
                                                        'S' : ['VDD'],
                                                        'B' : ['VDD'],
                                                        'layout_type' : 'series'     #shared_source, shared_drain, series, parallel
                                                     },

                          def define_derived_parameters(self) :
                              self.Wp = self.grid.snap(self.PNratio*self.Wn)
                '''

                # self.Wp = self.grid.snap(self.PNratio*self.Wn)

                self.pnrat_nand = 1.0
                self.pnrat_inv = 2.0
                self.vrailWidth = self.grid.snap(self.railWidth * 2.0)
                self.ps_PNratio = self.pnrat_inv
                self.num_stages = 3
                self.comparator_nmos_ratio = self.nmos_ratio  # > (unknown date) This doesn't need to match other cells so optimize it
                                                              #   to handle equal pmos and nmos finger widths (set to 0.5)
                                                              # > (1-24-13) If it doesn't match the ratio of the cells to the right then
                                                              #    it causes nwell to cut through the nmos diffusion region which ain't good

                self.r = 1.0
                self.number_one = 1
                self.number_five = 5
                self.nmos_ratio_pmos_cap = 0.24  # was 0.16 but reduced it for small cells
                self.vertical_pins = 'True'
                self.enable_deep_nwell = 'True'
                self.false_value = 'False'
                self.true_value = 'True'
                self.share_nwell_pmos_cap = 'False'
                self.extend_enclosing_layers_pmos_cap = 'False'

                # self.vertical_pitch_center = 4.0

                self.extra_vertical_pitch = 1.5

                # self.extra_vertical_pitch_vref = 1.5
                # self.DNW_spacing = 2.25

                self.volt_position_spacing = 1.44  # ideally the top 2 rows should be sharing an isolated PWELL but for the sake

                                                    # development time I am creating 2 isolated PWELL regions

                self.extra_nwell_space = 0.06  # TSMC DRC fix

                # self.extra_nov_spacing = 6.0

                self.extra_vertical_pitch_dnw = 3.4
                total_extra_spacing = self.volt_position_spacing \
                    + self.extra_vertical_pitch_dnw

                # calculate cell pitch based on desired cell height and fixed DNW_spacing parameter

                extra_ratio_for_nov = 1.0
                extra_ratio_for_volt_position = 0.7
                total_extra_raio = extra_ratio_for_nov \
                    + extra_ratio_for_volt_position
                self.verticalPitch = self.grid.snap((self.cell_height
                        - total_extra_spacing) / (len(self.layout_info)
                        + extra_ratio_for_nov))
                self.vertical_pitch_nov = self.grid.snap(self.verticalPitch
                        * (1 + extra_ratio_for_nov / 2.0))  # +self.extra_nov_spacing
                self.M4_rail_spacing = 0.16
                self.M4_signal_spacing = 0.04
                self.vertical_pitch_volt_position = \
                    self.grid.snap(self.verticalPitch * (1
                                   + extra_ratio_for_volt_position))
                self.vertical_pitch_caps = self.verticalPitch \
                    - self.extra_vertical_pitch


                # pdb.set_trace()
```

Bibliography

[1] "International Technology Roadmap for Semiconductors." [Online]. Available: http://www.itrs.net/Links/2012ITRS/Home2012.htm. [Accessed: 04-Dec-2013].

[2] D. James, "Inside the Apple A7 from the iPhone 5s," *Chipworks inside technology*. [Online]. Available: http://www.chipworks.com/en/technical-competitive-analysis/resources/blog/inside-the-a7/. [Accessed: 05-Dec-2013].

[3] J. Crossley, E. Naviasky, and E. Alon, "An energy-efficient ring-oscillator digital PLL," in *2010 IEEE Custom Integrated Circuits Conference (CICC)*, 2010, pp. 1–4.

[4] E. Alon, "Measurement and regulation of on-chip power supply noise," Stanford University, 2006.

[5] G. G. E. Gielen and R. A. Rutenbar, "Computer-aided design of analog and mixed-signal integrated circuits," *Proc. IEEE*, vol. 88, no. 12, pp. 1825–1854, 2000.

[6] M. G. R. Degrauwe, O. Nys, E. Dijkstra, J. Rijmenants, S. Bitz, B. L. A. G. Goffart, E. A. Vittoz, S. Cserveny, C. Meixenberger, G. Van Der Stappen, and H. J. Oguey, "IDAC: an interactive design tool for analog CMOS circuits," *IEEE J. Solid-State Circuits*, vol. 22, no. 6, pp. 1106–1116, 1987.

[7] R. Harjani, R. A. Rutenbar, and L. R. Carley, "OASYS: a framework for analog circuit synthesis," *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.*, vol. 8, no. 12, pp. 1247–1266, 1989.

[8] U. Choudhury and A. Sangiovanni-Vincentelli, "Automatic generation of parasitic constraints for performance-constrained physical design of analog circuits," *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.*, vol. 12, no. 2, pp. 208–224, 1993.

[9] G. Van der Plas, G. Debyser, F. Leyn, K. Lampaert, J. Vandenbussche, G. G. E. Gielen, W. Sansen, P. Veselinovic, and D. Leenarts, "AMGIE-A synthesis environment for CMOS analog integrated circuits," *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.*, vol. 20, no. 9, pp. 1037–1058, 2001.

[10] R. A. Rutenbar, G. G. E. Gielen, and J. Roychowdhury, "Hierarchical Modeling, Optimization, and Synthesis for System-Level Analog and RF Designs," *Proc. IEEE*, vol. 95, no. 3, pp. 640–669, 2007.

[11] M. Horowitz, M. Jeeradit, F. Lau, S. Liao, B. Lim, and J. Mao, "Fortifying analog models with equivalence checking and coverage analysis," in *2010 47th ACM/IEEE Design Automation Conference (DAC)*, 2010, pp. 425–430.

[12] P. Nuzzo, A. Sangiovanni-Vincentelli, X. Sun, and A. Puggelli, "Methodology for the Design of Analog Integrated Interfaces Using Contracts," *IEEE Sens. J.*, vol. 12, no. 12, pp. 3329–3345, 2012.

[13] *Virtuoso Analog Design Environment GXL*. Cadence Design Systems Inc.

[14] J. Rumbaugh, I. Jacobson, and G. Booch, "The unified modeling language reference manual," 1999.

[15] J. Crossley, "BAG Source Code." [Online]. Available: http://www.eecs.berkeley.edu/ ←crossley/BAG.

[16] L. Torvalds and J. Hamano, "Git: Fast version control system," *URL Httpgit-Scm Com*, 2010.

[17] R. E. Johnson and B. Foote, "Designing reusable classes," *J. Object-Oriented Program.*, vol. 1, no. 2, pp. 22–35, 1988.

[18] P. Coad and E. Yourdon, "Object-oriented design," 1991.

[19] J. Rumbaugh, M. Blaha, W. Lorensen, F. Eddy, and W. Premerlani, *Object-Oriented Modeling and Design*. Prentice-Hall, 1990.

[20] D. Liu, "A FRAMEWORK FOR DESIGNING REUSABLE ANALOG CIRCUITS," STANFORD UNIVERSITY, 2003.

[21] L. Prechelt, "An empirical comparison of seven programming languages," *Computer*, vol. 33, no. 10, pp. 23–29, 2000.

[22] "NumPy." [Online]. Available: http://www.numpy.org/. [Accessed: 24-Nov-2013].

[23] "SciPy." [Online]. Available: http://scipy.org/. [Accessed: 24-Nov-2013].

[24] "matplotlib." [Online]. Available: http://matplotlib.org. [Accessed: 24-Nov-2013].

[25] "A Python-embedded modeling language for optimization problems," *GitHub*. [Online]. Available: https://github.com/cvxgrp/cvxpy. [Accessed: 05-Dec-2013].

[26] "IPython Interactive Computing." [Online]. Available: http://ipython.org/. [Accessed: 24-Nov-2013].

[27] *MATLAB*. Natick, Massachusetts, United States: The MathWorks, Inc.

[28] "IPL Interoperable PDK Libraries." [Online]. Available: http://www.iplnow.com/index.php. [Accessed: 22-Nov-2013].

[29] "OpenAccess Coalition," *Si2*. [Online]. Available: http://www.si2.org/oac_index.php. [Accessed: 05-Dec-2013].

[30] D. M. Beazley and others, "SWIG: An easy to use tool for integrating scripting languages with C and C++," in *Proceedings of the 4th USENIX Tcl/Tk workshop*, 1996, pp. 129–139.

[31] N. Spurrier, "Pexpect," *Pexpect - Pure Python Expect-like module*. [Online]. Available: http://sourceforge.net/projects/pexpect/files/. [Accessed: 22-Nov-2013].

[32] "LSI Python for OpenAccess," *Si2*. [Online]. Available: http://www.si2.org/openeda.si2.org/projects/python4oa. [Accessed: 05-Dec-2013].

[33] "oaScript Extension Language Bindings," *Si2*. [Online]. Available: http://www.si2.org/openeda.si2.org/projects/oascript/. [Accessed: 05-Dec-2013].

[34] "YAML: YAML Ain't Markup Language," *YAML 1.2*. [Online]. Available: www.yaml.org. [Accessed: 05-Dec-2013].

[35] 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.

[36] I. E. Sutherland, R. F. Sproull, and D. F. Harris, *Logical Effort: Designing Fast CMOS Circuits*. Morgan Kaufmann, 1999.

[37] R. Nancollas, "Fully Automatic Standard Cell Creation in an Analog Generator Framework," EECS Department, University of California, Berkeley, 2013.

[38] *Helix*. Mountain View, CA: Synopsys.

[39] "Unity Analog Router." [Online]. Available: http://www.pulsic.com/products/pulsic-implementation-solution/unity-analog-router/. [Accessed: 24-Nov-2013].

[40] H.-P. Le, J. Crossley, S. R. Sanders, and E. Alon, "A sub-ns response fully integrated battery-connected switched-capacitor voltage regulator delivering 0.19W/mm2 at 73% efficiency," in *Solid-State Circuits Conference Digest of Technical Papers (ISSCC), 2013 IEEE International*, 2013, pp. 372–373.

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

[42] M. D. Seeman, "A Design Methodology for Switched-Capacitor DC-DC Converters," May 2009.

[43] W. Kim, M. S. Gupta, G.-Y. Wei, and D. Brooks, "System level analysis of fast, per-core DVFS using on-chip switching regulators," in *IEEE 14th International Symposium on High Performance Computer Architecture, 2008. HPCA 2008*, 2008, pp. 123–134.