

Fully Automatic Standard Cell Creation in an Analog Generator Framework

Rachel Nancollas

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2013-71

<http://www.eecs.berkeley.edu/Pubs/TechRpts/2013/EECS-2013-71.html>

May 15, 2013



Copyright © 2013, by the author(s).
All rights reserved.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission.

Acknowledgement

Elad Alon, John Crossley, Matt Spencer, Alberto Pugelli, members of the EEIS group, Elliott Sprehn, Tina, Michael and Philip Nancollas.

Fully Automatic Standard Cell Creation in an Analog Generator Framework

Rachel Nancollas

May 15, 2013

**Fully Automatic Standard Cell Creation in an Analog Generator
Framework**

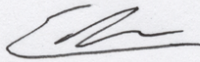
by Rachel Nancollas

Research Project

Submitted to the Department of Electrical Engineering and Computer Sciences, University of California at Berkeley, in partial satisfaction of the requirements for the degree of **Master of Science, Plan II**.

Approval for the Report and Comprehensive Examination:

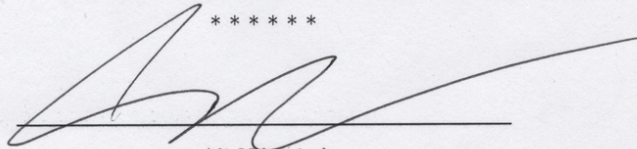
Committee:



Elad Alon
Research Advisor

5/14/13

Date



Ali Niknejad
Second Reader

5/14/13

Date

Fully Automatic Standard Cell Creation in an Analog Generator Framework
by Rachel Nancollas

Research Project

Submitted to the Department of Electrical Engineering and Computer Sciences, University of California at Berkeley, in partial satisfaction of the requirements for the degree of **Master of Science, Plan II**.

Approval for the Report and Comprehensive Examination:

Committee:

Elad Alon
Research Advisor

Date

* * * * *

Ali Niknejad
Second Reader

Date

Contents

1	Acknowledgements	7
2	Introduction	9
2.1	History of Synthesis	9
2.2	Problems with Automatic Analog Layout	9
2.3	Berkeley Analog Generator	10
2.4	Focus of this Work	10
3	Overview of BAG Code base	11
3.1	Schematic Sizing	11
3.1.1	Motivation	11
3.1.2	How to Use Sizing Routines	14
3.1.3	Overview of Sizing Classes	16
3.1.4	Walkthrough of Code in Sizing Example	17
3.2	Layout Information	19
3.2.1	Motivation	19
3.2.2	How to Generate Layout	19
3.2.3	Layout Modules	23
3.2.4	Layout Example	24
4	Automatic Debugging Tool	25
4.1	Motivation	25
4.2	Features of the Tool	26
4.2.1	Hierarchical Function Calls	26
4.2.2	Human-Readable Data Display	26
4.2.3	Clickable Layout Display	27
4.2.4	Layout Step-Through	27
4.2.5	Entire Layout or Snapshot Option	27
4.2.6	Decorator Syntax	27
4.3	How the Debugging Tool Works	27
4.3.1	Decorator: @debugwrapper	27
4.3.2	Printmeth	28
4.3.3	Savemeth	29
4.3.4	Snapshot Tool	30
4.4	Contribution	30
5	Routing Improvements	32
5.1	Motivation	32
5.2	Existing Standard Cell Routing Routines	33
5.3	Problems with Existing Routing Routines	34
5.4	Approaches to Fixing Vertical Route Intersections	35

5.4.1	Ordering of Routing Channels	35
5.4.2	Adding Additional Routing Channels	37
5.4.3	Removing Unused Routing Channels	39
5.5	Contribution	39
6	Automatic Standard Cell Generation	41
6.1	Motivation	41
6.2	Creating Layout Without Diffusion Breaks	42
6.3	Finding Euler Paths	44
6.3.1	Algorithms	44
6.3.2	Converting a Circuit to a Graph	45
6.3.3	Case 1: Consistent Euler Path	46
6.3.4	Case 2: Different Euler Path in PUN and PDN	47
6.3.5	Case 3: No Euler Path in either the PUN or PDN	48
6.3.6	Case 4: Euler Path in Devices of Different Widths	48
6.4	Mechanics of Moving from Schematic to Layout	48
6.5	Contribution	51
7	Conclusion and Future Work	52
7.1	Future Work	52
A	BAG Sizing Routines	53
B	BAG Layout Routines	56

List of Figures

1	Three types of BAG users and their roles.	13
2	BAG design flow for a generator writer.	13
3	Unsize, parameterized amplifier using BAG primitive devices.	15
4	Parameterized gain bandwidth testbench for BAG amplifier.	15
5	Instantiating a Design Module Object.	17
6	Sizing class hierarchy. Blue = DesignDescription, Purple=Interface, Pink=Workbench	18
7	Block diagram of standard cell layout block including code format.	24
8	Diagram showing how routing channels prevent intersections with horizontal and vertical metal layers of different types	24
9	Diagram showing inheritance of main layout modules.	24
10	Left - Debugger: Hierarchical method calls. Right - Debugger: Data structure display.	28
11	Left - Debugger: Clickable layout with geometry labels. Right - Debugger: Tool for stepping through the layout.	29
12	Diagram showing the main purpose of the debug methods as well as their helper functions.	31

13	Intermediate steps in the creation of layout of a standard cell captured by the layout debugger.	32
14	Two standard cells demonstrating the three basic types of routes: M1, M2, and poly as well as the joined poly connection.	34
15	Diagram showing the difference between pins, terms, instPins, and instTerms.	35
16	Standard cell showing the problem of vertical routes intersecting	36
17	Diagram showing how vertical routes can intersect.	36
18	The three basic techniques for removing intersections in standard cell routing: (1) reordering bars, (2) duplicating all the bars, (3) using a hybrid of both techniques to add minimum numbers of bars.	37
19	Diagram showing the steps of the algorithm that orders the routing channels to produce the smallest number of shorts.	38
20	Layout on left shows an intersection. Layout on right shows intersection being removed by adding an extra routing channel.	40
21	Layout on left shows a space violation. Layout on right shows the violation being corrected by adding an extra routing channel.	40
22	Layout before and after unused bars are removed.	41
23	Layout of the boolean network on the left in the current implementation (top) and proposed diffusion-breakless implementation (bottom).	42
24	This figure shows the removal of edges from a graph as an Euler path is being built. Variables related to the pseudo code are also displayed.	45
25	Conversion of a circuit schematic to a graph for Euler path identification.	46
26	Automatically generated layout with the same Euler path in the PUN and PDN. The schematic on the left shows the Euler path found by the modified Fleury's algorithm.	47
27	Automatically generated layout with no complete Euler paths. The PUN does not meet the Euler condition while the PDN has gates of different sizes.	49
28	Automatically generated layout with a diffusion break on the top and a complete Euler path on the bottom because the bottom device has been split into two devices of the same width.	50

1 Acknowledgements

Engineering at its best is a team activity. In this vein, there are many people who helped make this work and my (hopefully) subsequent graduation possible. First, I would like to thank my advisor Elad Alon for his knowledgeable advice, patience, and understanding. I would also like to thank John Crossley who developed BAG and spent many hours helping me debug and understand his code. I also appreciate all the discussion and support from my many colleagues both in BWRC and other research groups. In particular, I would like to acknowledge Matt Spencer for his willingness to discuss research problems even when he was busy taping out every few months and Alberto Puggelli for all his advice while writing this thesis.

I would also like to thank Elliott Sprehn for his love, moral support, constant encouragement, and enthusiasm about my research. In spite of working on real software, he still thinks BAG is awesome and I appreciate that. He also gets special props for explaining how to use GIT (about six times). I feel lucky to have found someone who inspires me intellectually and personally, and I hope our journey together is long. I would also like to thank my family for supporting me through all of my education, while also for encouraging me to make decisions that made me happy. I would not be where I am today without your guidance, support, and love. Extra thanks to my mom for sending the world's best care packages!

Abstract

Automated analog integrated circuit design has been an outstanding problem since the late 1980s. One of the reasons for this is there are few tools for helping designers create the standard cell libraries used by most analog synthesis software. The Berkeley Analog Generator (BAG) approaches the problem of automating analog layout by providing a software framework that aids designers in codifying their design process to create layout generators. The specific focus of this work is to take this automation one step further to enable complete automation from schematic level to circuit layout. This is done by creating special classes for fully automatic custom digital layout, improving routing algorithms to require less manual setup, and developing a debugging tool to help designers understand their automated layout. Several circuit examples are presented to demonstrate the capabilities of this complete analog automation.

2 Introduction

Integrated circuit layout is a notoriously complicated and detail-oriented process. Thus, the growth of circuit complexity quickly led to the development of CAD tools to aid designers in keeping track of design rules and verifying their layout. Due to the repeatability of many circuit elements, it was not long before the idea of codifying aspects of the layout process was born.

2.1 History of Synthesis

The birth of synthesis can be placed in 1986 when Dave Johansen presented the idea of taking user specifications and automatically generating integrated circuits at the 16th Design Automation Conference [1]. Although this may have been the first formal definition of synthesis, the ideas that make up the backbone of many modern synthesis techniques were in development as early as the 1960s. In 1961, Lee developed the “Lee maze router”, which is what most modern routers are based on [2]. A decade later, Hashimoto and Stevens published the idea of channel routing [2]. Although there have been many improvements to these algorithms and ideas, the basis of modern routing has its roots in the sixties. It was also during the 60s and 70s that IBM started using layout patterns to create regular arrays [2].

One of the key events that paved the way for Johansen’s idea and research on EDA at an academic and industry level was Mead and Conway’s book about how VLSI chip fabrication actually worked [3]. After this book was released in the early 1980s, ideas for layout tools flourished. Most of the ideas of this decade were based on graph theory, convex optimization, stochastic methods, genetic algorithms, and physical laws [3] [2]. The 1990s ushered in the area of extremely high clock rates and with it massive amounts of heat dissipation. In this decade, it became necessary to do thermal analysis of chips, so methods of solving heat equation PDEs were developed [3]. As device sizes began to shrink in the 2000s, the major innovation became working on schemes to lower leakage current [3]. Although issues of leakage and evolving CAD to work with developing technologies continue to be major EDA issues today, one of the most critical issues facing the CAD community is analog synthesis. In the 2009 NSF report on the future of EDA, analog and mixed signal design was listed as a “foundational area for future DA support” [3]. The report cites the two reasons that analog synthesis development is particularly important: (1) 66% of designs are AMS and (2) the analog portions of designs are the most likely to fail.

2.2 Problems with Automatic Analog Layout

The current dearth of good solutions for analog synthesis is not for lack of trying. Analog synthesizers have been in development since the late 1980s. In 1989, Harjani et al. published an oft-cited work on an analog framework they called OASYS [4]. This framework uses a library of functional blocks that contain detailed design and synthesis information. It then takes sized schematics and decomposes them into these functional blocks, which can then be synthesized. A decade later, Koza, et al. developed a analog synthesis technique that uses genetic programming [5]. Their idea was to take base circuits for what they refer to as the “eight problems of analog circuit synthesis”

and use genetic algorithms to modify these circuits by moving wires, inserting devices, assigning component parameters, and re-using chunks of other circuits [5]. Once again, this synthesis routine uses a library of functional blocks they call “embryo circuits” that are modified to create fully formed adult circuits. One of the best papers on this subject is Martins et al.’s paper on LAYGEN II [6]. This synthesis approach combines a number of tools and techniques to generate layout meeting user specified design constraints [7]. Specifically, a user selects the type of circuit to synthesize as well as the constraints. A topology selection tool chooses an appropriate topology from a library of function blocks. Once selected, the circuit is sized and verified, subblocks are synthesized and laid out, and parasitics are extracted so the circuit can be verified post-layout. This software package is a powerful example of how decades of research can be combined to make automated analog layout feasible. In spite of this progress, none of these tools have become mainstream in the analog design community. Analog automation companies such as Barcelona Design Inc. have failed in part because it is difficult to license analog IP blocks because technology changes quickly and it can be difficult to reuse those blocks. Without a clear model on how to make profits, commercial analog CAD companies have trouble continuing development. Moreover, even freely licensed tools developed in universities are rarely used. One reason for this lies in the fact that all of these synthesizers have, at the core, a library of generators that contain all the design information. Whether they are called functional blocks or embryo circuits, these base cells have to be created manually by expert designers and it can be hard to add additional cells to the library to extend the range of these tools.

2.3 Berkeley Analog Generator

The goal of the Berkeley Analog Generator (BAG) is to give designers tools create the code for these library cells [8]. Although BAG contains high level sizing, placement, and routing tools to combine these cells to create full SOC integrated circuits, it is also structured to make it easy for designers to incorporate their favorite tools to optimize sizing, routing, etc. For example, although the BAG sizing routines currently use Spectre simulations, there are classes that allow designers to plug in SPICE or other simulators instead. BAG’s major innovation is allowing designers to select common *styles* of layout and providing tools to help designers create parameterized, process independent generators. Incorporating cell creation into the analog synthesis design flow means a designer can truly create automated circuits from specifications without necessarily relying on existing functional blocks. The goal is that this will lead to wider adoption of automated analog layout in the IC community.

2.4 Focus of this Work

The specific focus of this thesis is to take this automation one step further to enable complete automation from schematic level to circuit layout. Currently, BAG has helper classes for sizing schematics and separate helper classes for layout. In between, the designer must fill out a layout template that specifies the backbone for the design that can then be laid out. This template differs for each style of layout and is fairly simple to fill in. The idea of the template is to help a designer capture the general layout thoughts she has about the design so the layout helper classes can do

the detailed work of figuring out where to place devices and routes. Although this template is simple, for many styles of layout, it is possible to create an initial template automatically. This thesis makes this possible for one specific style of layout: custom digital standard cells. In order to do this, three main pieces of code were developed:

- Improved routing methods to automatically fix shorts that sometimes occurred in the original standard cell router implementation
- A custom standard cell class that (1) extracts information from sized schematics to create a layout template and (2) figures out how to lay out devices and order routing channels
- A debugging tool for layout visualization to help designers (1) understand how the generator code actually works and (2) make minor modifications to the initial layout template after it has been automatically generated

The rest of this thesis describes these pieces of code. Specifically, section 3 describes the schematic sizing and layout routines in the BAG code base, with the purpose of giving some context for the code that was developed for this thesis. This section also serves as a reference for future BAG users about how to use the code base. The next section (section 4) discusses the debugging tool for layout visualization. This tool is also designed to help users understand how the code base works and how to debug layout errors. Once this groundwork is laid, section 6 details the code that fully automates standard cell creation. This is the main focus of this work. During the creation of these automatic layout routines, several problems with existing routing algorithms were discovered. Therefore, section 5 discusses improvements to these routines. Finally, section 7 presents some conclusions and future work

The overarching goals of this work are to show the success of BAG as an analog synthesis tool by showing some examples of fully automatic generators. The other main goal is to create a code framework for fully automatic standard cells that could be extended to other styles of layout. The next section will begin addressing these goals by giving an overview of the entire BAG code base to provide context for these fully automatic standard cells.

3 Overview of BAG Code base

3.1 Schematic Sizing

The BAG codebase consists of two smaller, unconnected code bases: a schematic sizing framework and a circuit layout framework. Each of these code bases provides helper functions to assist designers in codifying their design process. This section will provide details on what these helper functions are and how they work. First, the schematic sizing routines will be discussed.

3.1.1 Motivation

As any analog circuit designer knows, schematic sizing is an iterative, time consuming process. At one extreme, some circuit designers manually adjust parameters and run simulations until they

obtain the desired behavior. At the other end, some prefer to use complex optimization software to do design more automatically. A number of these programs rely on creating device and circuit behavior models using posynomials and attacking the problem with convex optimization to reduce runtime [9]. Other programs apply concepts such as evolutionary algorithms to enable arbitrary circuit/behavioral models and find local minima [10]. There have also been techniques that try to reduce the number of parameters being optimized by creating libraries of cells (such as OTAs) that are described by a simpler set of equations [11]. Although the results produced by this approach can seem less “magical” than solutions from a convex optimizer, the cell libraries must be created by expert designers, so it can be difficult to design new circuits.

It is this black-box reputation of fully automatic analog circuit optimizers that has prevented their widespread adoption. Instead, most modern designers use a combination of mathematical analysis to narrow the range of parameters and then a simulator (SPICE, Spectre, etc.) to test the design and find the exact value of most of the parameters. Mathematical approaches include a wide range of techniques including convex optimization and V^* analysis which uses the value of $\frac{2I_D}{gm}$ to find the device sizes which meet bandwidth and gain constraints. In addition to these calculations, many designers use a simulator like Spectre ADE both as a lookup table to map parameters such as gm to device widths and as a method for fine tuning their parameters. Some designers take this a step further and actually write their own scripts in Ocean to automatically run simulations. This idea of having “spice in the loop” is what motivates the schematic sizing portion of BAG.

Even the most die-hard GUI users likely have some sort of script to do schematic sizing. The problem is these scripts are all very individualized, often vary from project to project, and may be distributed (for example, a MATLAB script for calculating sizes and a Ocean script for running simulations). It is unlikely that another designer would choose to decipher this jumble of code to recycle the design or move it to another technology node. Therefore, the vision of BAG is to create a technology independent schematic sizing routine that allows a designer to use any process they wish. The simulator interface is wrapped inside a Python interface, which allows the designer to use any simulator or optimizer they choose. They can also use Python’s matplotlib for doing calculations and creating figures.¹ Essentially, BAG’s schematic sizing framework is an organized, consistent way for designers to write their sizing routines. Being able to rerun a designer’s code without having to learn a different interface should allow for easier design reuse, modification to meet new requirements, and transitions to different technology nodes. In fact, the circuit design scripts created in BAG are known as generators because the intent is that any designer could use a generator to create a circuit that meets their specifications. As such, there are three groups of people who might interact with BAG: developers who maintain BAG’s code base, generator writers who are circuit designers codifying their design process using BAG, and generator users who use generators to create circuits with a given set of specifications. These roles are shown in Fig. 1. Currently, the workflow that a BAG generator writer follows begins by creating a parameterized schematic in Cadence using process independent “BAG primitives”. A BAG testbench using the

¹John Hunter, matplotlib’s creator, died of cancer recently. If you’ve used John’s contributions to this project, consider donating: <http://numfocus.org/johnhunter/>

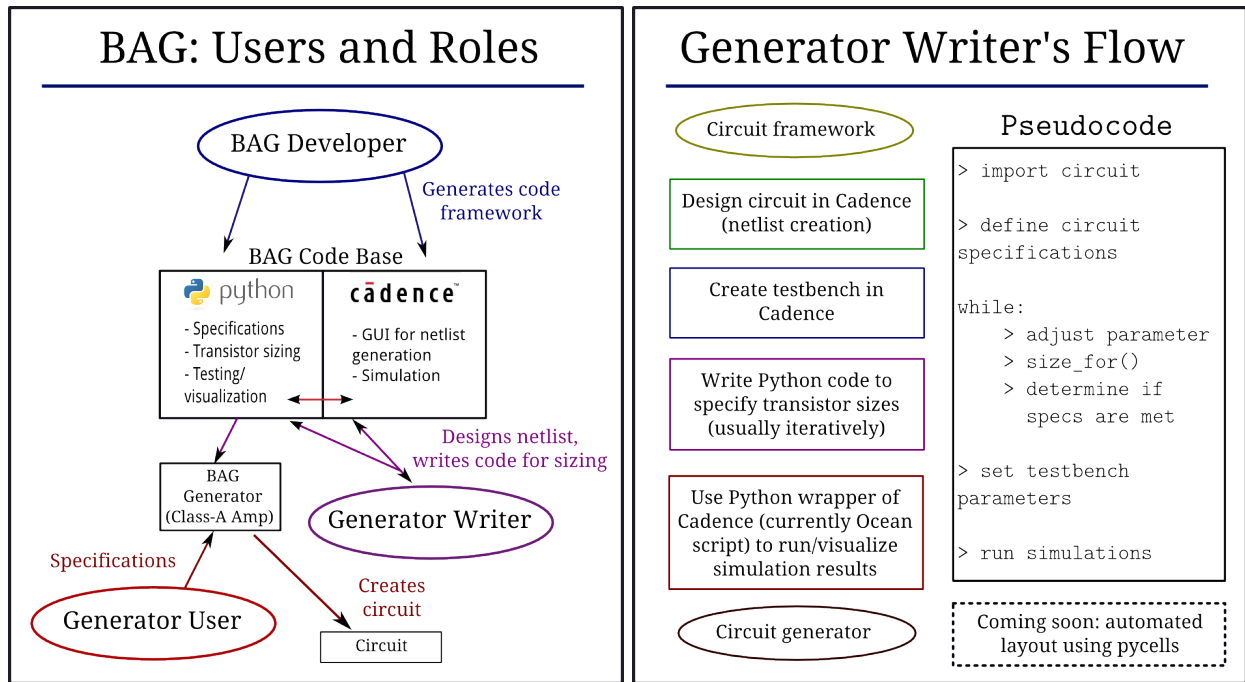


Figure 1: Three types of BAG users and their roles. Figure 2: BAG design flow for a generator writer.

parameterized schematic can also be created. These Cadence circuits are then read into a BAG Project which contains the netlist and parameters as Python objects. The user then writes code to modify the parameters and test the circuit using her design method of choice. In this process, she can use the *size_for* method to look up sizes of transistors based on the desired gm and I_D . Once she has selected parameters, these are inserted into an Ocean simulation using the Python wrapper. This simulation can be run in an open or closed loop to simply view results or to fine tune parameters. Matplotlib allows the simulation results to be processed and displayed. This workflow is shown in Figure 2.

3.1.2 How to Use Sizing Routines

To illustrate how the BAG sizing classes abstract the details of the Cadence interface from the user, the following code example is shown below.

```

1 # xxx used in place of circuit values for confidentiality
2 prj = BAG.BagProject()
3 prj.import_sch_to_template(cell='amp_ex1',lib_name='BAG_templates')
4 from BAG_templates__amp_ex1 import BAG_templates__amp_ex1 as amp_ex1
5
6 amp = amp_ex1(prj) # instantiates object of class
7 tb = amp.testbench_modules['BAG_templates__amp_ex1_tb_gain_bandwidth']
8
9 # Set some circuit values
10 tb.vddval = xxx # set all testbench parameters...
11 tb.set_parameters('W',xxx) #this function sets all parameters named 'W'
12
13 # Sizing loop
14 for itail in arange(0,xxx,xxx) :
15     NM12_sizer = amp.NM1.size_for(corner='tt',vstar=0.2,ibias=itail/2,vds=xxx,
16     lookup_only=True,unit_width=unit_width)
17
18     for i in range(1,10) :
19         prev_mult = -1
20         NM12_vds_target = round(vout_com-(vin_com-NM12_sizer['vgs']),2)
21         NM12_sizer = amp.NM1.size_for(corner='tt',vstar=0.2,
22         ibias=itail/2,vds=NM12_vds_target,lookup_only=True,unit_width=unit_width)
23
24         if NM12_sizer['mult'] == prev_mult :
25             break
26         prev_mult = NM12_sizer['mult']
27
28 # All remaining transistors are sized

```

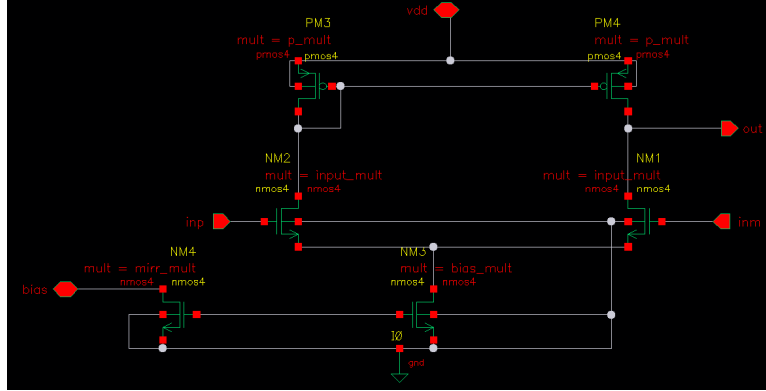



Figure 3: Unsized, parameterized amplifier using BAG primitive devices.

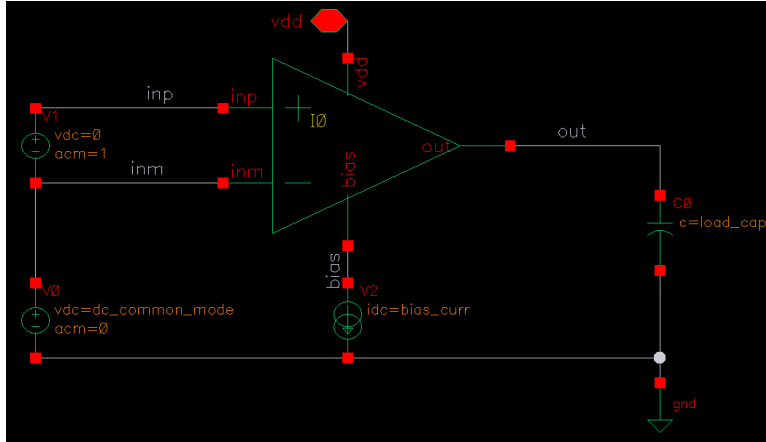


Figure 4: Parameterized gain bandwidth testbench for BAG amplifier.

```

29     NM34_sizer = amp.NM3.size_for(corner='tt',vstar=0.2,ibias=itail,
30     vds=round((vin_com-NM12_sizer['vgs']),2),unit_width=unit_width)
31
32     # Run simulation
33     tb.run_sim()

```

In this example, a common source amplifier with a pMOS current mirror is created in Cadence (Figure 3). The Cadence schematic has BAG primitive mosfets and parameterized multipliers for the bias, mirror, and input transistors. A gain-bandwidth testbench is created to test this schematic (Figure 4). This testbench also has parameterized values for VDD, the DC common mode, and the bias current. Once the schematic and its associated testbench(s) are created in Cadence, they have to be read into BAG. This happens in lines 2-7 of the code snippet above. The details of how the BAG-Cadence interface works and how schematic objects are created in BAG is described in section 3.1.4.

Once a BAG project is created, the user creates variables to represent circuit parameters. Parameters in the testbench are set directly by accessing them with *tb.parameter = value*. Parameters that appear in the subcircuit are set using the **tb.set.parameters** syntax as shown in lines 9-11.

At this point, the circuit designer still needs to set the multipliers for all the transistors in order to run the gain-bandwidth simulation. There are many different goals a designer could have when choosing the multipliers, but in this case, she is attempting to set a common mode output voltage. To do this, she sweeps the tail current and for each of these tail current values, she sizes the input transistor to get the **mult** value that corresponds to a given V^* and ibias value at some guess of Vds (line 15-16). Using this initial sizing, she now has an initial guess for Vgs. With this Vgs value, she can calculate her target Vds for the transistor based on her ideal Vout and Vin. Specifically, $Vds_{target} = V_{out,CM} - (V_{in,CM} - Vgs_{estimated})$ (line 20). Her next step is to re-size the transistor using her Vds_{target} to get closer to the actual transistor multiplier. This is done in a loop, which terminates when the Vds_{target} is close enough so the multiplier stops changing (lines 18-26). Once the optimal sizing is determined for the input transistors, the **size_for** function is used on the additional transistors using the known bias current (lines 28-30). These sizes are set so the testbench simulation can be run and the gain-bandwidth results can be plotted (lines 32-33).

3.1.3 Overview of Sizing Classes

In order to explain how this script actually works in BAG, it's useful to look at the classes that make up the BAG codebase. Figure 6 shows the inheritance of the most relevant classes in the BAG sizing modules. The blue circles represent classes that are part of the DesignDescription module, the purple ones are part of the Interface module, and the pink ones can be found in the Workbench module. Roughly, the DesignDescription module describes an abstract representation of circuits in BAG, the Interface module contains classes that interface with simulators to allow circuit testing, and the Workbench module has classes that provide data processing and device sizing functionality.

In particular, the DesignDescription module has a number of classes that represent circuits from the primitive device level to the unit cell level as well as the testbench level. It also contains classes that represent properties of circuits such as design parameters and netlists. The Interface module has classes that allow the user to test circuits in Spectre and other simulators. An outline describing the main BAG classes and their functionalities can be found in Appendix A.

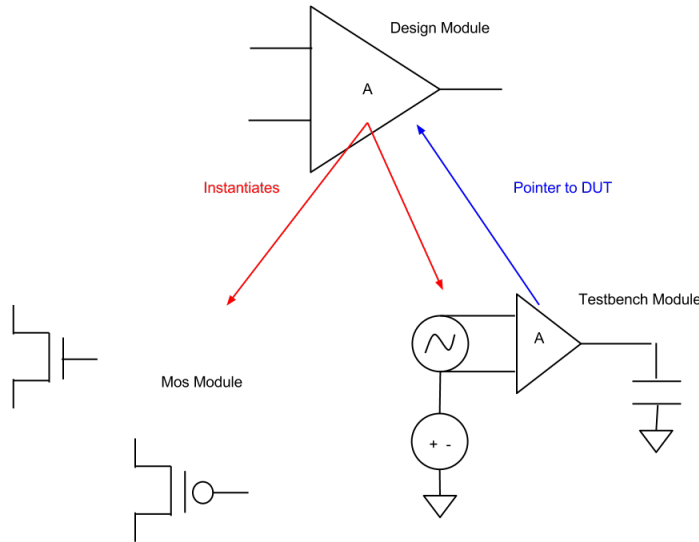


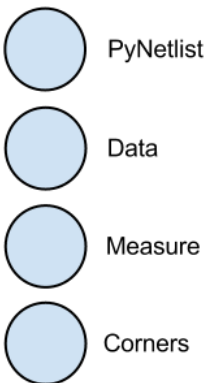
Figure 5: Instantiating a Design Module Object.

3.1.4 Walkthrough of Code in Sizing Example

To further explain how these sizing classes interact, this section describes what happens in BAG during each method call in the amplifier code sample above.

Creating a BAG Project The process starts by creating a BAG Project (line 2) which stores all parameters about the BAG session and a hook to the virtuoso session. When the `import_sch_to_template` method is run (line 3), the BAG project launches an ocean script that reads the schematic into a yaml file and then parses that information into a BAG Template. Line 4 actually imports the template class that is created and line 6 creates an amplifier object that is linked to the BAG Project.

Creating an Amplifier Object When the amplifier object is created, there are a number of things that are initialized. First, pins and connections are copied from the yaml file. Also, based on netlist information, parameters are created and submodules (either subcircuits or devices) are instantiated. Any existing testbenches are also instantiated as shown in Fig. 5 and linked in. In line 6, the testbench connected with the amplifier is assigned to a variable for ease of calling.



Sizing Sizes are set explicitly in lines 10-11. Parameters can either be set at the testbench level or for the entire circuit (using `set_parameters`). In the sizing loop, the `size_for` function is used as a lookup table to find the size of a transistor with the given set of properties. Currently, this actually just calls a lookup table, but in the future, it could connect to a simulator. Sizes are then set using the output of the sizing routines in lines 29-30.

18

extracted from the testbench probes so it can be processed and displayed.

3.2 Layout Information

3.2.1 Motivation

The Layout module is one of the key motivations behind BAG. As discussed, one of the key challenges of analog layout is its sensitivity to matching, parasitics, and process variation. Matching is particularly important and designers have been working to model its impact on circuit performance since the late 1980s [12] [13]. In digital synthesis, circuits tend to have significant noise margins, so the lack of matching caused by “spaghetti” layout is not as problematic. There have been some attempts to produce automated layout that is constrained by symmetry and matching requirements, but it has not been adopted [14] [15]. Due to its sensitivity, analog layout is often considered an art, and designers tend to be skeptical of letting a machine take over this process. Another factor that prevents adoption is most analog engineers are not programmers and have already spent many years customizing their CAD tools. Thus they are hesitant to learn to use another complex layout tool, especially during a time sensitive tapeout.

BAG has several features to make it more attractive to circuit designers. First, it gives designers control over the layout process in that a designer must think about the general layout structure she wants and then write code to produce this layout. This produces layout that is easy to read in comparison to “spaghetti” layout generated by typical digital synthesis tools. At its core, BAG gives designers tools to codify the process they have already been using to reduce the tedium of pushing polygons and make it easier to modify/reuse layout. Although initially doing layout in BAG may not save much time, the ability to perform instant re-sizing and reuse blocks in different technology nodes makes the investment worthwhile.

Another feature of BAG is that instead of a complex and specific design languages (such as Skill), its code base and layout templates are all written in python. The choice of Python was motivated by the fact that it is easy to learn and extremely well documented (a search for writing a for-loop in Python brings up 1.4 million results while a similar search in Skill produces only 265). Moreover, Ciranova (now owned by Synopsis) has a large code base of PyCells which provide methods to create, size, and move layout geometry. BAG uses these methods to create a hierarchical class structure that allows users to easily describe and compile their layout.

3.2.2 How to Generate Layout

Layout generation begins when the designer creates a layout template such as the one shown below. This code inherits from the `std_cell` (standard cell) class and can be compiled to create layout for a Nand gate. The first part of the code is a constructor for the Nand object. It defines lists of parameters, layout structure, and parameter values. These parameters contain information about any variable names and values such as width, length, size ratio information, etc. Most importantly, there is a `layout_info` data structure that both defines the netlist information and also contains in-

formation about layout structure. Before describing the contents of this data structure, it is useful to discuss the format of a standard cell (there are some other layout styles: standard block, array, and eventually amplifier, but these will not be discussed here). A standard cell has a rectangular form factor, as shown in Figure 7. Specifically, the top and bottom are power rails, below and above there are device groups, and finally in the middle are routing channels. Roughly, routing channels are in M1 and device terminals are connected with vertical M2 or poly bars as shown in Fig. 8. This is done to avoid intersection (although as will be discussed in Section 5, there are some additional checks needed to remove shorts).

The elements of **layout_info** are as follows. The first and last element of **layout_info** contain information about the power rails such as layer, implant information, and terminal information. The next element(s) contain information about pull up (pMOS) device groups. A device group is defined by a name, the type of device it is, the net labels of its terminals, its layout (series, parallel, shared source, shared drain), and some placement information. This is actually the original style of layout, section 6 will describe the new **custom_std_cell** class that moves away from specifying series, parallel, etc. The device element also has information about how parameters are mapped. In this example, the width (“W”) parameter of the pMOS devices is described by “Wp”. There is also an option to add dummies around device groups, but this is not shown here. In the middle of this list is a routing group element. This contains information about each routing channel such as name, layer, and terminal. Below this are pull down (nMOS) device groups.

Below the **layout_info** data structure is a default dictionary that contains default values for the parameters described in the paramNames dictionary as well as some other parameters set in parent classes. The next part of the code overrides the setupParameters method in **std.cell** and its parent class **BAG_pycell**. The purpose of this method is to add the parameters that have been defined above to the newly instantiated pycell object. Before the parameters can actually be added to the **mySpecs** dictionary, they are prepared so they meet design rule constraints. For example, in the case of “L”, the minimum length is found in the tech file (line 96). This value is then snapped to the layout grid (line 97). A step constraint is created so the length is only set to values allowed by the design rules (lines 98-99), and the length value is finally added to the **mySpecs** dictionary in line 100. This is done for all the parameters and the **mySpecs** is added into the **specs** dictionary using the name mapping defined in **cls.paramNames** (line 107).

A final (optional) step is to add parameters that depend on the values of other parameters. For example, in this circuit **Wp** (the width of the pMOS device) is defined as the product of **Wn** (the width of an nMOS device) and **pnRatio** (the ratio of a pMOS device to an nMOS device). Overloading **define_derived_parameters** allows the user to add these defined parameters so if the nMOS width is changed, the pMOS width will change accordingly. Once this constructor has been written and these classes are overridden, it is possible to compile the PyCell. It is important to note that compiling this PyCell creates a device with default sizes and that different sizes can be passed into the layout if it is instantiated by a higher level circuit.

```

1 from __future__ import with_statement
2 from BAG_layout import *
3 from operator import itemgetter
4 import inspect
5
6 class Nand( std_cell ) :
7     paramNames = dict(
8         L           = 'L',
9         Wn          = "Wn",
10        Wp           = "Wp",
11        use_widths   = "use_widths",
12        PNratio      = 'PNratio',
13        stack_2n     = 'stack_2n',
14    )
15
16    layout_info = [
17        { 'name' : 'VDD',
18          'type' : 'rail',
19          'layer' : 'metal1',
20          'terminal' : 'VDD',
21          'term_type' : "INPUT_OUTPUT",
22          'contact_to' : 'diffusion',
23          'enclose_contact' : ['nimplant', 'nwell']
24        },
25
26        { 'name' : 'p_row1',
27          'type' : 'device_group',
28          'tran_type' : 'pmos',
29          'height_ratio' : None,
30          'vertical_align' : "SOUTH",
31          'subelements' : [ { 'name' : 'Mpmos',
32                             'param_mapping' : {'W' : 'Wp', 'L' : 'L' },
33                             'G' : ['IN1', 'IN2'],
34                             'D' : ['OUT'],
35                             'S' : ['VDD'],
36                             'B' : ['VDD'],
37                             'layout_type' : 'parallel',
38                         },
39
40                    ]
41        },
42    ]

```

```

43 { 'name' : 'route_group1',
44     'type' : 'route_group',
45     'subelements' : [{ 'name': 'OUT', 'layer': 'metal1', 'terminal': 'OUT', ...
46                         'term_type': "OUTPUT" },
47                        { 'name': 'IN1', 'layer': 'metal1', 'terminal': 'IN1', ...
48                          'term_type': "INPUT" },
49                        { 'name': 'IN2', 'layer': 'metal1', 'terminal': 'IN2', ...
50                          'term_type': "INPUT" },
51                      ]
52 },
53
54 { 'name' : 'n_row1',
55     'type' : 'device_group',
56     'tran_type': 'nmos',
57     'height_ratio' : None,
58     'vertical_align' : "NORTH",
59     'subelements' : [ { 'name': 'Mnmos',
60                         'param_mapping': { 'W': 'Wn_calculated', 'L': 'L' },
61                         'G' : ['IN1', 'IN2'],
62                         'D' : ['OUT'],
63                         'S' : ['GND'],
64                         'B' : ['GND'],
65                         'layout_type' : 'series'
66                      },
67                    ]
68 },
69
70 { 'name' : 'GND',
71     'type': 'rail',
72     'layer': 'metal1',
73     'terminal' : 'GND',
74     'term_type': "INPUT_OUTPUT",
75     'contact_to': 'diffusion',
76     'enclose_contact': ['pimplant']
77 },
78 ]
79
80 default = dict(Wn=1.0, Wp=2.0, nmos_ratio=0.36, PNRatio=2.0, ...
81               verticalPitch=5.2, extra_rail_space_bottom=0.12, use_widths='False')
82
83 #####
84

```



```

85  @classmethod
86  @debugwrapper
87  def defineParamSpecs( cls, specs):
88
89      # Super makes all the parent functions available in the new defineParamSpecs
90      super(Nand, cls).defineParamSpecs(specs)
91      mySpecs      = ParamSpecArray()
92
93      # Some specs are left out for brevity
94
95      # Set up specs (including constraints) and add them to the dictionary
96      L = specs.tech.getMosfetParams( 'nmos', cls.oxide, "minLength") # minimum length
97      L = cls.grid.snap( L) # snap to grid
98      stepConstraint = StepConstraint( cls.maskgrid, start=L, ...
99          resolution=cls.resolution, action=FailAction.REJECT)
100     mySpecs("L", cls.default.get('L',L), constraint = stepConstraint)
101
102     PNratio = cls.default.get("PNratio",2.0)
103     mySpecs("PNratio",PNratio,constraint=...
104         StepConstraint(0.01,start=0.2,action=FailAction.REJECT))
105
106     # Parameter renaming: adds mySpecs (what you've defined above) into specs
107     renameParams( mySpecs, specs, cls.paramNames)
108
109     #####
110
111  @debugwrapper
112  def define_derived_parameters(self) :
113
114      self.Wp = self.grid.snap(self.PNratio*self.Wn)
115      self.Wn_calculated = self.grid.snap(self.Wn*self.stack_2n)

```

3.2.3 Layout Modules

To understand how this layout template is converted into a pycell during the compiling step, it is useful to examine some of the major layout modules shown in Fig. 9. The key layout modules include **std.cell** and its children classes that provide helper functions for creating several styles of layout. The parent class for **std.cell**, **RoutingMethods**, contains methods for properly laying out routing channels and connecting standard cells. Finally, the primitive PyCell classes have helper functions for creating primitive devices such as resistors and transistors. More details about these classes and their methods can be found in Appendix B.

Layout Information

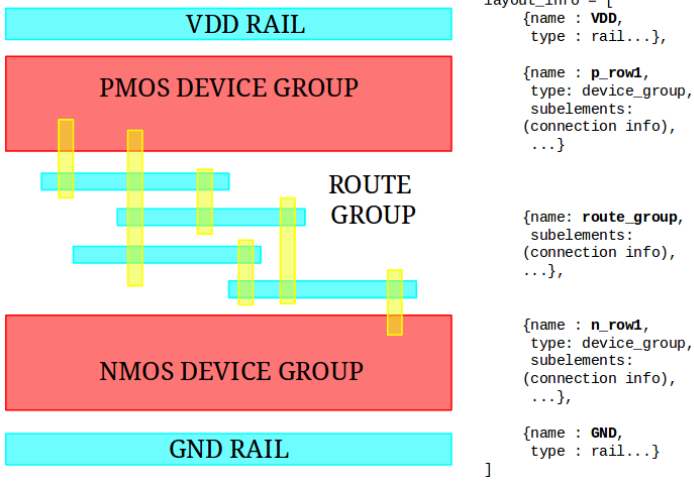


Figure 7: Block diagram of standard cell layout block including code format.

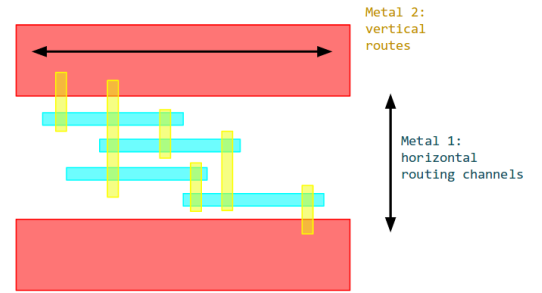


Figure 8: Diagram showing how routing channels prevent intersections with horizontal and vertical metal layers of different types

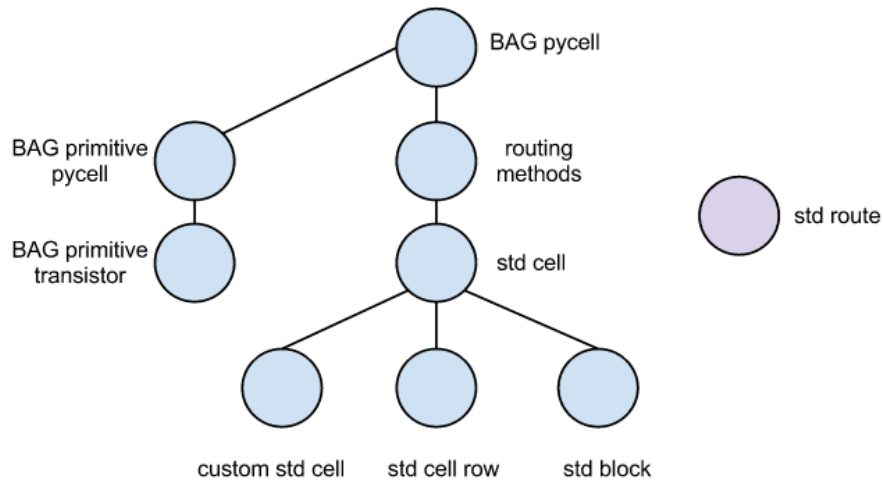


Figure 9: Diagram showing inheritance of main layout modules.

3.2.4 Layout Example

When a standard cell (such as the Nand PyCell shown in the code sample above) is compiled, the Ciranova compiler calls the following methods. (These are taken directly from the Ciranova

Python API Documentation: [16]) First, **defineParamSpecs** is called which creates a dictionary of parameter names for the layout. Next, **setupParams** adds these parameters to the object and also does some initial layout calculations. During **genTopology**, the devices, rails, and routing channels are created, but not placed in the correct orientation. **sizeDevices** is the next method that is called. It simply sizes the transistors based on their length, width, and height constraints. Finally, **genLayout** is called. During this routine, routes, rails, and devices are placed. The devices are also resized and the number of fingers is recalculated. This is also the step where routing connections are made. Specifically, devices are connected to bars with metal or poly routes and adjustments are made to these routes so they are DRC compatible. Finally, a few clean-up steps are preformed to complete the layout.

4 Automatic Debugging Tool

4.1 Motivation

One of the challenges of using the layout tools in BAG is that it can be difficult to troubleshoot problems. From the description of how the layout code is executed, an important observation is that when the PyCell is compiled, the Ciranova compiler looks for the five overridden methods (listed in section 3.2.4) and executes these. Unfortunately, this makes it difficult to manually override this process and just run a single method or part of a method. With judicious use of flow control statements, it is possible to only run a section of code, but this is complicated and can require extensive modifications. Moreover, if only a section of code is run, this can raise exceptions in other methods. If exceptions are raised, then the layout cannot be viewed. These two problems make it very difficult to see the layout as it is being generated.

In addition to seeing the layout for debugging purposes, this tool is also useful for understanding how layout works. Specifically, Ciranova's layout methods involve a lot of creation and movement of geometry that is difficult to keep track of without images. For example, consider trying to imagine how fifty rectangles change from one step to another by just reading their coordinates. Therefore, there is utility in having a tool that allows the user to view how the layout changes after each method call.

Another feature that is missing from current debugging techniques is being able to set breakpoints and see the layout. It is often desirable to set a breakpoint, access some of the data structures to see what is being modified, and also look at the layout. Available debugging tools do not allow the layout to be opened until it has been compiled. This debugger saves a snapshot of the layout each time a method is called, which makes it easy to view incomplete geometry. An unexpected utility of this is occasionally Ciranova will throw a segfault for unknown reasons. Typical errors throw some sort of stack trace, so it is possible to track the error to some line of the code. Obviously, segfaults provide no such luxury, so the only way to find the error is using a binary search of break statements. With the debugger, it is possible to run the code and figure out based on the last saved file where the code unexpectedly stopped.

In addition to seeing how the layout is formed, the fact that Ciranova calls only five methods (which in turn call hundreds of other methods in the BAG code base) makes it difficult to follow the stack trace. Therefore, this debugging tool prints a hierarchal list of the methods in then order of when they are called and what methods call them.

Finally, although it is easier to visualize than layout, it is also difficult to access data structures from one method call to another. With a complex code base such as BAG, most data structures are many layers deep. Without knowing where parameters are stored, it can be difficult to find this information. For example, the default value for length is stored in:

```
self.device_type_yaml[parameters_limits]['L']['default_value']
```

To address these problems, a debugging tool was developed that displays the names and contents of data structures in a human-readable format and only prints data when it has been changed.

4.2 Features of the Tool

The features of the debugger are designed to address the problems discussed above and are listed here. Running the debugger on a standard cell requires the following steps:

1. in `../PDK/pycell_work/Debug` run the RUNONCE script the first time the debugger is run
2. in `../PDK/pycell_work/BAG_layout/BAG_pycell` change the **flag** variable to True
3. compile the code and in a browser open
`file://[directory path]/PDK/pycell_work/htmldir/[NameofCell]/home.html`

To run the debugger on a non-standard cell, refer to `../PDK/pycell_work/Debug/README`. When the debugger is run, it creates a locally stored website that contains information about the layout as it is created.

4.2.1 Hierarchal Function Calls

On the homepage is a list of all the method calls in a hierarchal fashion as shown in Fig. 10 (left). An indented method indicates that it was called by the method one indentation level above it.

4.2.2 Human-Readable Data Display

When a method is clicked on, it opens a page that has information about what happens during that method call. Specifically, it lists all the fields in the standard cell namespace as shown in Fig. 10 (right). The data is listed with links to the top of the page. Dictionaries (which can be long and cumbersome to scroll through) are accessible with hyperlinks and it is possible to return to the top of the page by clicking on the dictionary. Since users are typically interested in what changes from one method call to the next, information is only printed if the data fields have changed.

4.2.3 Clickable Layout Display

In addition to displaying data fields, the most important feature of the debugger is the layout images. Any method that contains layout information has an image of the layout with clickable geometry. Mousing over the geometry changes the color of the selected rectangle and clicking on it displays its name and coordinates as shown in Fig. 11 (left).

4.2.4 Layout Step-Through

Often it is useful to be able to see the progression of the layout without the data fields. Therefore, the homepage has an option to just see the layout. This allows the user to move forward or backward through the layout and access the clickable layout information at each method call. An example of this layout display is shown in Fig. 11 (right).

4.2.5 Entire Layout or Snapshot Option

Depending on the type of problem a user is solving, she might want to see the progression through the entire layout (as described above) or she might only be interested in how layout changes between two places in the code. Therefore, the debugger tool can either be used by changing the flag in **BAG_pycell** and recording information about the entire layout process or by adding a `snapshot(self)` line anywhere in the code. The webpage created by the snapshot option is saved in `file://[directory path]/PDK/pycell.work/snapshotdir/[NameofCell]/home.html`. Recording information about the entire layout increases run time rather significantly so it is only recommended for standard cells. The snapshot tool is better for larger cells.

4.2.6 Decorator Syntax

Decorators are used to record methods in the debugger. The debugger syntax is just `@debugwrapper` right above the `def` statement.

4.3 How the Debugging Tool Works

This section describes the mechanics behind the key features of the debugging tool. It is accompanied by Fig. 13 which shows the main purposes of the debugging methods and the helper functions they call.

4.3.1 Decorator: `@debugwrapper`

When the layout debugger homepage is opened for a particular standard cell, it displays a hierarchal list of all methods called during the creation of the cell. This is created using Python's stack inspection tool. Specifically, when a method is called during debugging, the decorator method (**`debugwrapper`**) is called. This method inspects the stack and makes a list of the current function being called and all the functions that called it. This list is passed to the first main debugging function (**`printmeth`**).

Method Call for Creating a Nand

- [defineParamSpecs](#)
 - [defineParamSpecs](#)
 - [defineParamSpecs](#)
 - [_compute_contact_spacing](#)
- [setupParams](#)
 - [setupParams](#)
 - [updateProperties](#)
 - [define_derived_parameters](#)
 - [initialize_layout_info](#)
 - [initialize_fingers](#)
 - [initialize_height_constraints](#)
 - [initialize_fingers](#)
 - [increment_fingers](#)
- [genTopology](#)
 - [create_rail](#)
 - [createRoutingChannel](#)
 - [defineParamSpecs](#)
 - [defineParamSpecs](#)
 - [defineParamSpecs](#)
 - [defineParamSpecs](#)
 - [setupParams](#)
 - [setupParams](#)
 - [setupParams](#)
 - [setupParams](#)
 - [updateProperties](#)
 - [genTopology](#)
 - [add_dummy](#)
 - [genLayout](#)
 - [add_pins](#)
 - [layout_dummy](#)
 - [setupParams](#)
 - [setupParams](#)
 - [setupParams](#)

Hierarchical Method Display

```

-
device_type
  type 'str'
  pch_lvt
result_file
  type 'str'
dummies_right
  type 'int'
  0
params_from_file
  type 'NoneType'
  None
true_mult
  type 'str'
  1
device_intent
  type 'str'
  fast
param_file
  type 'str'

```

[Top](#)

Layout Information

[Top](#)

Contents of params

- W : 200e-9
- dummy_side : top
- dummies_left : 0
- L : 60e-9
- pin_out_file :
- device_intent : fast

Data Fields During Method Call

Figure 10: Left - Debugger: Hierarchal method calls. Right - Debugger: Data structure display.

4.3.2 Printmeth

Within **printmeth**, a unique html page is created for that method call (if **create_rail** was called multiple times, the instances would be called **create_rail0.html**, **create_rail1.html**, etc.). In addition, the name of the page and the list of methods are saved in a timestamped file. Using the timestamped file that was created in the previous method call, **printmeth** is able to examine the list of method calls to determine the indentation of the current method name on the homepage. For example, if it determines that **createRoutingChannel** was called by **create_rail** then **createRoutingChannel** has one more level of indentation than **create_rail**. Finally, a copy of all of the fields in the standard cell object at that point in time are saved to the timestamped file. This occurs in the method **saveself** and is done so that the data can be compared in later methods to determine what has changed. This copy of the data structures as well as the location of the html page is returned at the end of **printmeth**.

After these data structures are returned, the **debugwrapper** calls the main function that it is

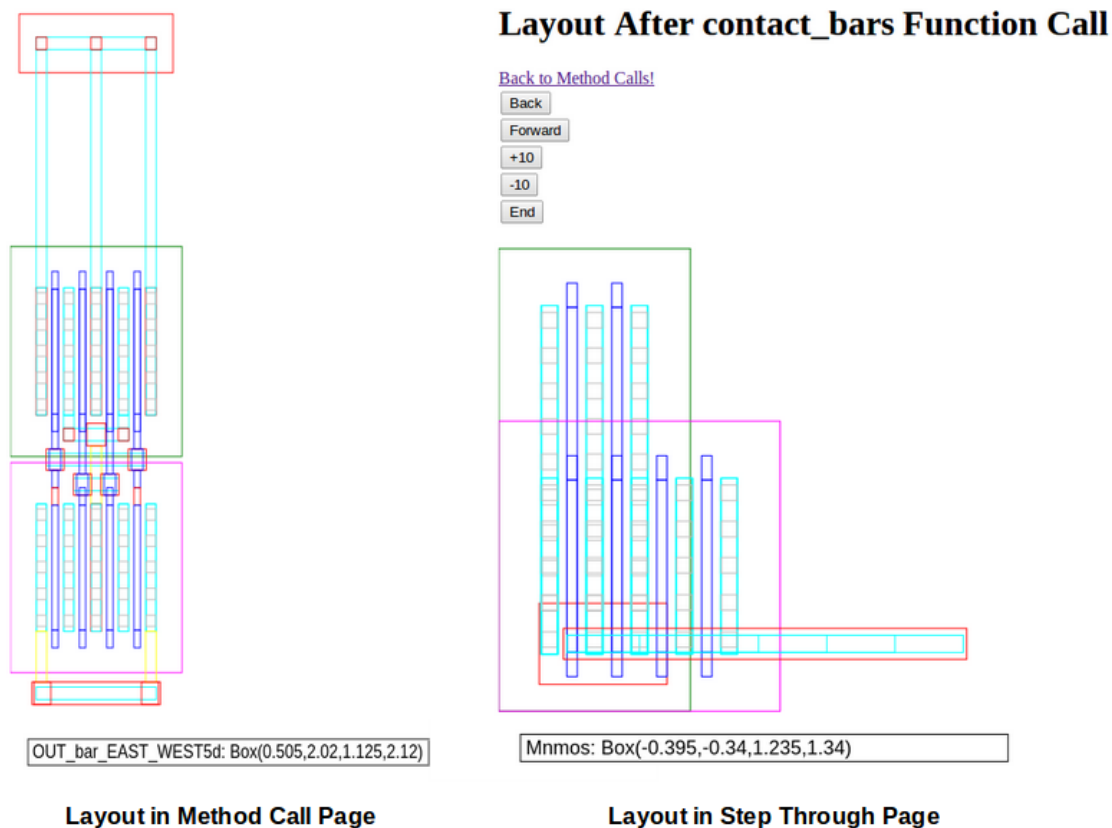


Figure 11: Left - Debugger: Clickable layout with geometry labels. Right - Debugger: Tool for stepping through the layout.

decorating. Finally, it uses the data structures from **printmeth** as arguments to call **savemeth**.

4.3.3 Savemeth

During **savemeth**, two types of pages are created: (1) pages that are linked from the hierarchical homepage and contain data field and layout information and (2) pages that just show layout information that can be stepped through sequentially. Figure 11 (right) shows the progression of layout while stepping through the debugger. The layout specific pages contain a picture of the layout with its fields and have buttons for navigation through the layout generation. The method specific pages from **printmeth** are filled in with the contents of any data fields of the standard cell object that have changed during the method call. The differences are determined by comparing to the data structure copy passed in from **printmeth**. These pages also contain a picture of the layout.

The layout pictures are to scale and are colored to match the layer. Also, when moused over, the geometry turns black, and when it is clicked on, the name of the geometry and its dimensions

appear in the box below the diagram. The interactive clicking functionality is implemented using javascript. Specifically, a function called **getcoords** is called that goes through all of the geometry in the standard cell and makes a list of the coordinates, layer, and name of the element. Some of this processing is done in **procelement** and **procother**. Once these lists are created, **drawstuff** is called. This method does some processing of the coordinate, name, and layer lists to create the x and y coordinates for the rectangles (**cordplot**) and the color of the rectangle based on the layer (**colorcheck**). This information is then printed in the html file as a javascript array. At the end of **drawstuff**, a javascript page is copied in below the array that contains the methods that allow for mouse overs.

The method specific pages also contain information about data fields. On the page for the first method call, all of the information contained in the data fields is printed. The data is processed when **proccself** is run. On subsequent pages, only data fields that have changed since the last method call are displayed. To do this, the data field information that was saved in **printmeth** is compared to current data field information generated by **saveself**. The information that is distinct is put into dictionaries. Additionally, a layout diagram is generated using the procedure described above and put in a temporary html file. At this point, **printall** is called. This method prints the distinct data and also copies the layout information in the temporary file to a canvas. This process is repeated for all methods marked with the decorator syntax. At the end of this process, it is possible to go to the homepage and access information about standard cell fields at any method call during the compilation of the PyCell.

4.3.4 Snapshot Tool

The snapshot tool is almost identical to the debugging tool. The main exception is that it can be inserted anywhere in the code and takes a snapshot of the layout and fields at that point rather than at the end of the method call.

4.4 Contribution

Currently, the two main CAD companies that produce software for generating designer-driven analog layout are Cadence and Synopsys. Cadence uses pCells which are written in Skill. Currently, Cadence does not have an interactive method for viewing layout. In order to view partially completed layout, it is necessary to modify the pCell to include breakpoints and other pass statements. However, if the pCell fails to compile, partially created layout cannot be viewed. This makes it difficult to view the layout that is created before an error occurs. On the other hand, Synopsys (which acquired Ciranova) uses PyCells. PyCell Studio includes a tool called **cnexp** or Ciranova Explore that allows users to interact with layout. Specifically, users can execute line(s) of code and see the resulting layout. This is a useful tool for debugging, but it lacks several features which the debugger here provides. Specifically, stepping through the code in this manner is very time consuming. While it is good if the problem has been narrowed down to a few lines of code, it is difficult to get a high level view of what is going on to figure out where the error has occurred. In addition, it does not allow the user to access the data structures. Although this can be done by

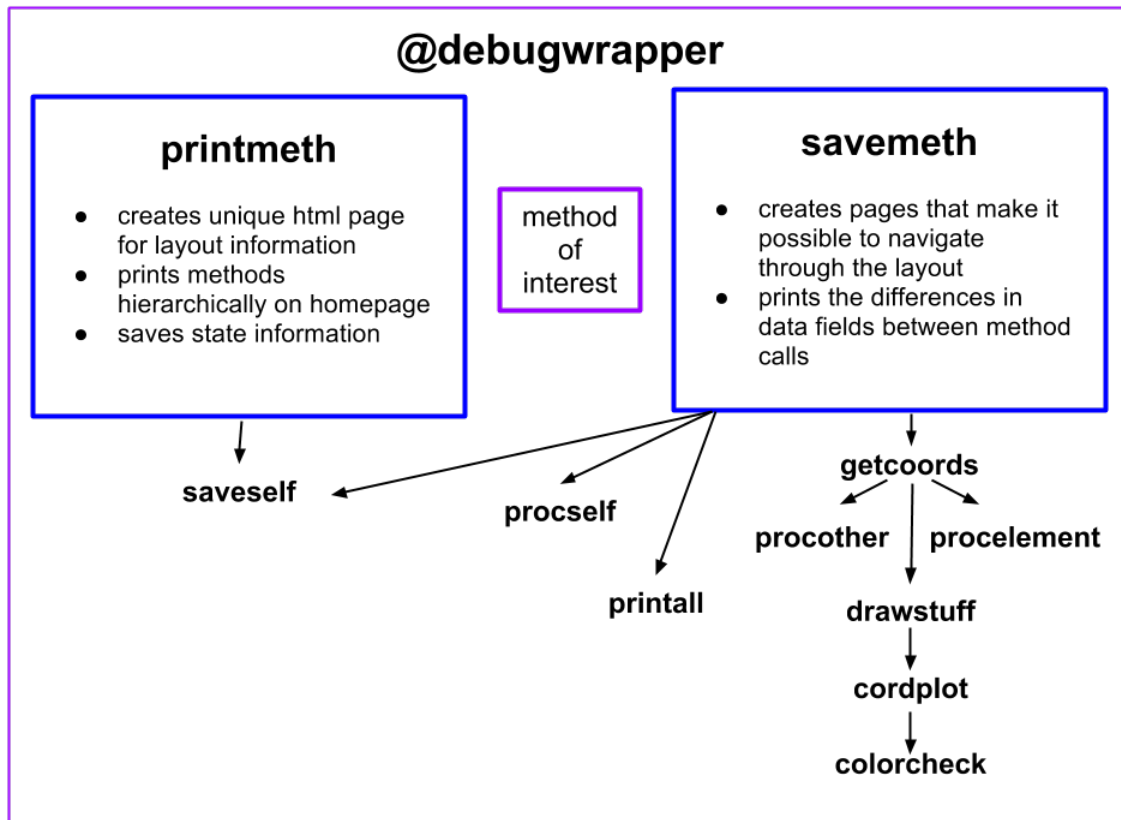


Figure 12: Diagram showing the main purpose of the debug methods as well as their helper functions.

putting breakpoints in the code, it is tedious to access a particular field (especially if the name of the field is unknown).

This debugging tool provides several innovations which are useful for debugging and more importantly helping new users understand the code base. These include (1) viewing the hierarchy of method calls, (2) viewing all PyCell fields at any given point in the code, (3) viewing the layout at any step in the code, and (4) being able to step through the entire layout creation quickly to pinpoint an error. Moreover, unlike other debugging tools which have the specific function of finding an isolated error, this tool was initially developed to teach the author about the code base. Given the resistance of many analog designers to switching to analog generators such as BAG, having more tools like this that provide a view into the inner workings of a complex code base may help analog CAD gain acceptance in the IC community.

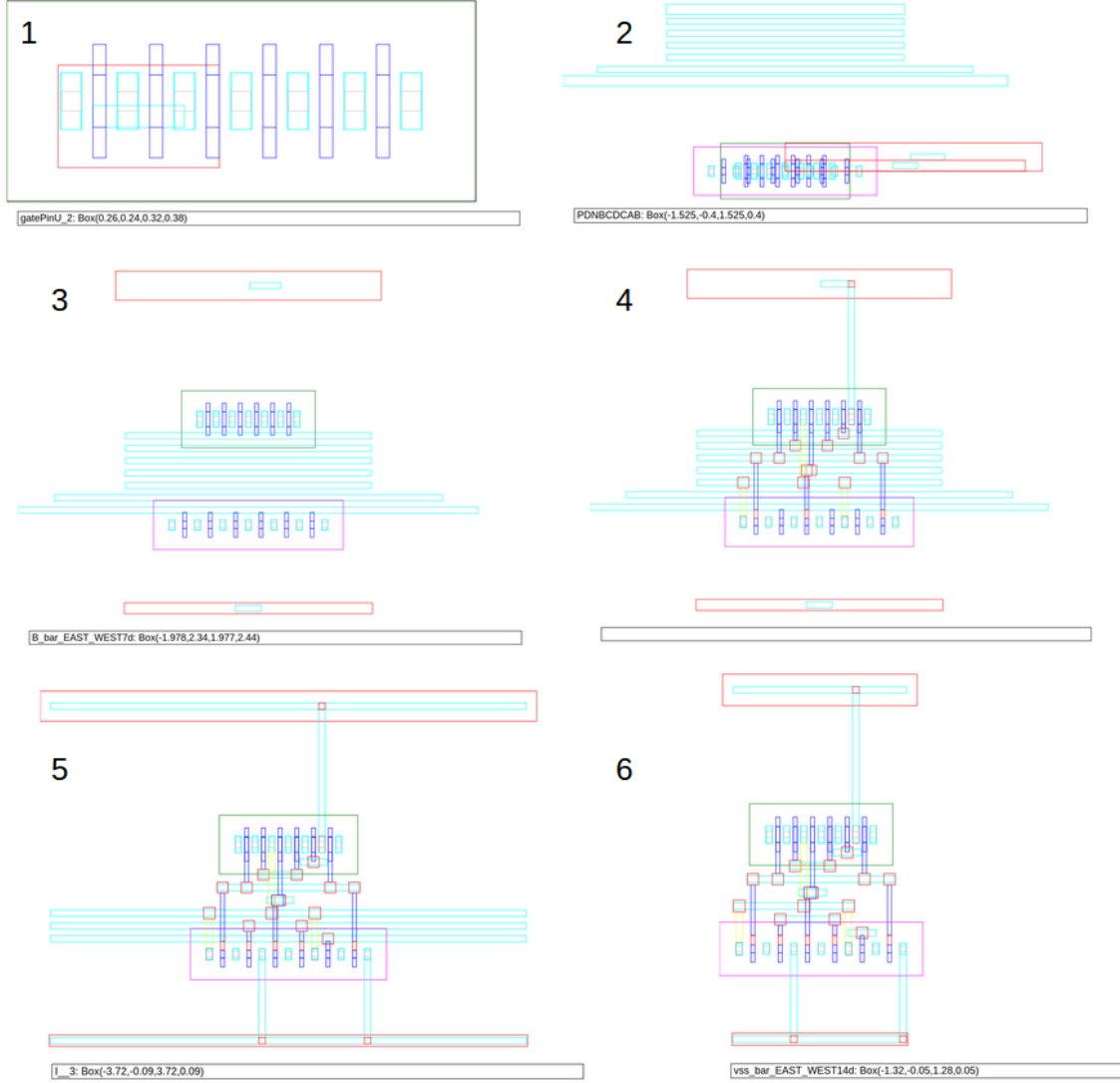


Figure 13: Intermediate steps in the creation of layout of a standard cell captured by the layout debugger.

5 Routing Improvements

5.1 Motivation

As anyone who has done manual layout will attest, routing is one of the most frustrating parts of the process. Finding ways to connect layers while keeping routes short and obeying design rules is tricky. Moreover, without careful initial planning, it is easy to run into a situation where it is impossible to make a final connection and large sections of routing must be redone. A problem

like this with specific geometric constraints and a need for pre-planned paths seems well suited for algorithmic design. In fact, in the late seventies and eighties, many papers were published outlining a variety of algorithms for automatic routing. In 1977, Bruer published a paper that described a placement method based on minimum cut algorithms rather than purely on distance placement [17]. In the eighties, Rivest et al. described a router that used a grid-based layout format and greedy algorithms to create routing channels [18]. One of the most cited papers published at that time was Yoshimura et al.’s work on a variety of routing algorithms based on graph theory [19]. It was during this time that researchers and CAD companies began creating automatic digital synthesis tools. With the development of digital synthesis tools, it became obvious that many of the efficient tools for digital routing were not suited for the symmetry and parasitic constraints of analog layout. Therefore, in the next ten years, analog routers such as KORAN and ANAGRAM II were published by Cohn et al. [20], but did not catch on. Robust analog routing is an outstanding problem and solutions such as Martin et al.’s evolutionary approach to placement and routing continue to be published as late as 2012 [6].

In the future, it may be advantageous to incorporate some of these advanced routing algorithms into BAG. At this stage in the project, it was decided that it made more sense to use standard cells with simple routing methods for several reasons. First, although standard cells result in somewhat less dense routing, the routing routines are much easier to write and understand. Also, as technology nodes get smaller, design rules will likely require routing to be grid based, so this sort of regular cell routing will likely become common for analog design. The following section will describe the existing standard cell routing routines and their problems as well as detail improvements to these routines.

5.2 Existing Standard Cell Routing Routines

Standard cells are a fairly common way to do layout and are particularly well suited for the custom digital/mixed signal circuits that are the focus of this thesis. As shown in Fig. 7 and 8, the basic idea behind a standard cell is to make a “layout sandwich” with power rails and device groups on the outside and routing channels in the middle. Each net that requires an input/output pin or connects devices requires a horizontal routing channel. At the standard cell level, these horizontal channels are in metal 1 and are placed between the device groups. Specifically, when layout is created, each routing channel specified in the `layout_info` data structure has a M1 bar associated with it. These bars are placed between the pMOS and nMOS devices with enough space so a contact can be placed on adjacent bars without violating design rules.

Next, vertical routes must be made between instPins on the device groups and the routing bar of the same net. (See Fig. 15 for a depiction on the difference between pins, terminals, instPins, and instTerms.) This is done in the **straight_line_instpin_to_bar** method. There are three types of connections that can be made. If a connection is being made between the drain or source of a device and a route bar that is adjacent to the device group, then the route can be made in M1 because it will not intersect other horizontal M1 routing bars. However, if the route bar is not adjacent to the device group, the route must be made in M2 so it does not short out routing bars

between it and the device. Finally, if the connection is made between a gate and a route bar, the route is done in poly. These three types of connections are shown in *noEuler Test Gate* in Fig. 14. Once these routes are created, there is also a check if two poly bars connect to the same bar and intersect each other. If this happens, one of the contacts is removed and the bars are joined so they do not violate design rules. An example of this happening is shown in *Nand* in Fig. 14. Once the vertical connections are made, to reduce parasitics, horizontal bars are shortened so they only extend to the pins they connect. There is an option to extend bars to the right or left for the purposes of joining standard cells, but this option is not used in this work.

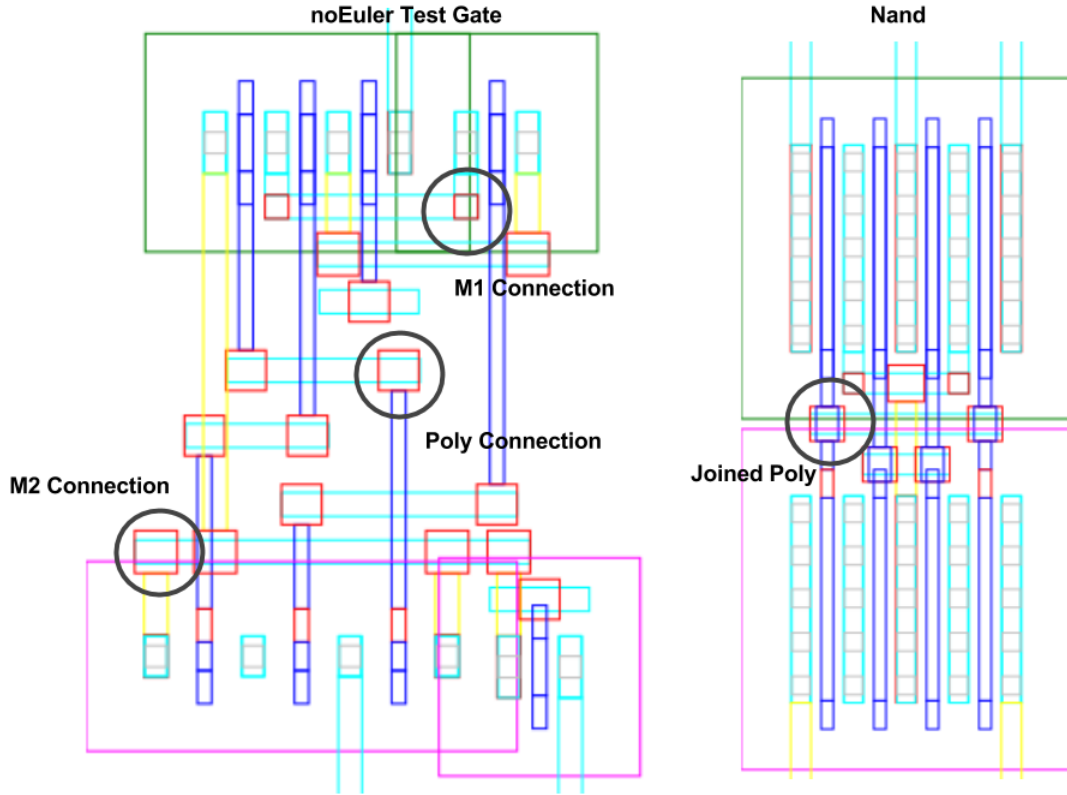


Figure 14: Two standard cells demonstrating the three basic types of routes: M1, M2, and poly as well as the joined poly connection.

5.3 Problems with Existing Routing Routines

One of the problems with the original routing implementation is there are no checks to make sure bars being routed south from the pull-up network (PUN) do not intersect with bars being routed north from the pull-down network (PDN). An example of this problem is shown in Fig. 16.

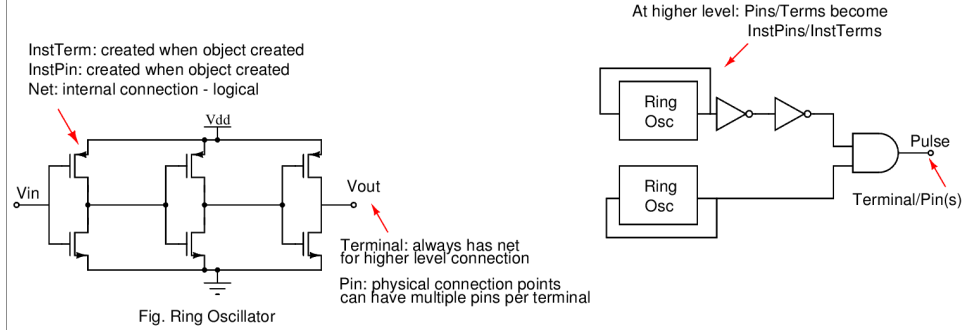


Figure 15: Diagram showing the difference between pins, terms, instPins, and instTerms.

Specifically, if a route from the PUN connects to a bar that is farther south than a bar accessed by the PDN, these routes can intersect if they are horizontally close to each other. The problem is most easily visualized with the diagram in 17. In this image, the vertical route from the **A** gate in the PUN intersects with the vertical route from the **B** gate in the PDN because of the placement of the routing channels. In fact, in this particular standard cell, there are two intersections noted by the black “X”. As hinted by the figure, there are several ways to mitigate this problem which will be the subject of the next section.

5.4 Approaches to Fixing Vertical Route Intersections

In order to fix intersections in vertical routing and still use the standard cell routing channel layout, there are two main things that can be done. First, the order of the bars can be changed as shown in Fig. 18 [1]. Compared to the same cell in Fig. 17, just swapping the order of bars “C” and “A”, reduced the number of intersections. Second, instead of bothering to reorder, all of the bars can be duplicated as shown in Fig. 18 [2]. Here, the PUN and PDN connect to different routing channels, which are later connected (i.e. C0 to C1) on the side of the cell or in another layer. This is the simplest approach, but has the most area cost. These solutions can be combined so that the bars are reordered to remove as many shorts as possible, as shown in Fig. 18 [3]. In this case, the bars were reordered to remove one short and then another routing channel was added to remove the second intersection. Writing an algorithm to do this automatically was the focus of the improvements made to the routing routines.

5.4.1 Ordering of Routing Channels

One of the challenges of ordering the routing channels to avoid vertical intersections is the decision about where bars are placed is made before layout geometry is produced. As discussed above, the first step in creating layout is to create a **layout_info** data structure that contains, among other things, the order of the routing channels. Depending on the horizontal location of the pins of the PUN and PDN, this order could cause intersections, but at the point when it is assigned, no layout exists so it is impossible to tell. Therefore in order to decide how to order the routing bars, this algorithm is based on the assumption that all the gates of the PUN and PDN line up (like in Fig.

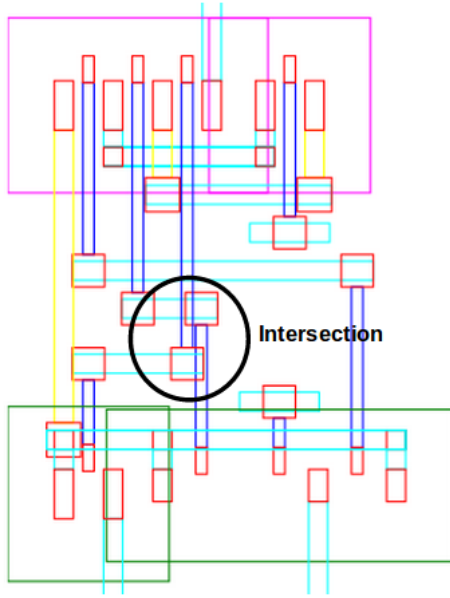


Figure 16: Standard cell showing the problem of vertical routes intersecting

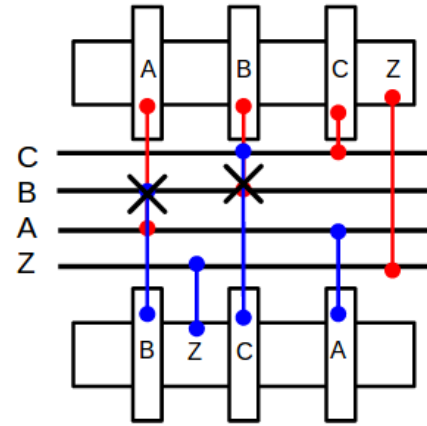


Figure 17: Diagram showing how vertical routes can intersect.

17). As will be shown later, this assumption does not always hold, but in some cases, it may result in fewer shorts.

The algorithm starts with an ordered list of the gates in the PUN (`PUNgates`) and PDN (`PDNgates`) as well as an empty list of the order of routing channels (`routeList`). Pseudo-code describing the algorithm is shown below. To explain the code consider Fig. 19. In stage 1, the first two gates are considered: the PUN gate is added to the top of the list and the PDN gate is added to the bottom. In stage 2, the PDN gate is not in `routeList`, so it is added to the bottom. In stage 3, both gates are in the `routeList` so nothing happens. Finally, in stage 4, the gates are the same (and not already in the `routeList`), so “D” is added to the middle of the list.

```
while PUNgates and PDNgates have unpopped terms:
```

```
    if PUNgates[0] AND PDNgates[0] is already in routeList:
        do nothing to routeList
```

```
    elif PUNgates[0] == PDNgates[0]:
        put the common term in the middle of routeList
```

```
    elif PUNgates[0] AND PDNgates[0] are both not in routeList:
        add PUNgate to top of list and PDN gate to bottom
```

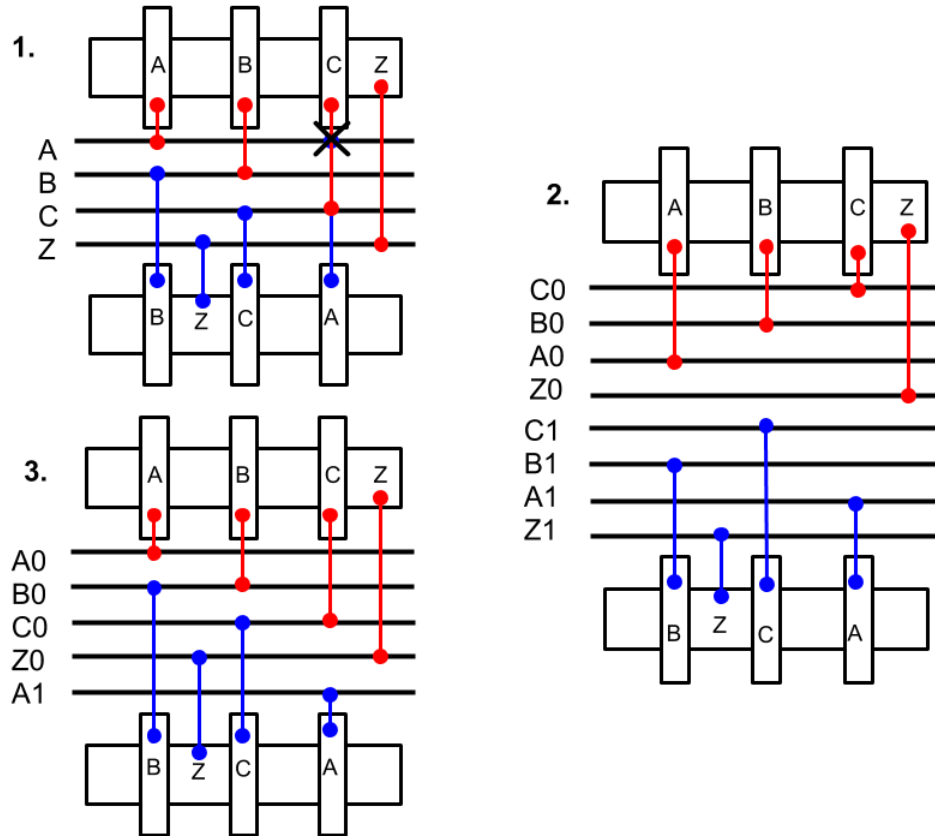


Figure 18: The three basic techniques for removing intersections in standard cell routing: (1) reordering bars, (2) duplicating all the bars, (3) using a hybrid of both techniques to add minimum numbers of bars.

```

elif PUNgates[0] is not in routeList:
    put PDNgates[0] at routeList[end]

elif PDNgates[0] is not in routeList:
    put PUNgates[0] at routeList[0]

pop first term of PUNgates and PDNgates

```

5.4.2 Adding Additional Routing Channels

Once the bars have been ordered, layout geometry gets created as the Ciranova compiler calls `genTopology` and `genLayout`. During the calls of `straight_line_instpin_to_bar` when vertical

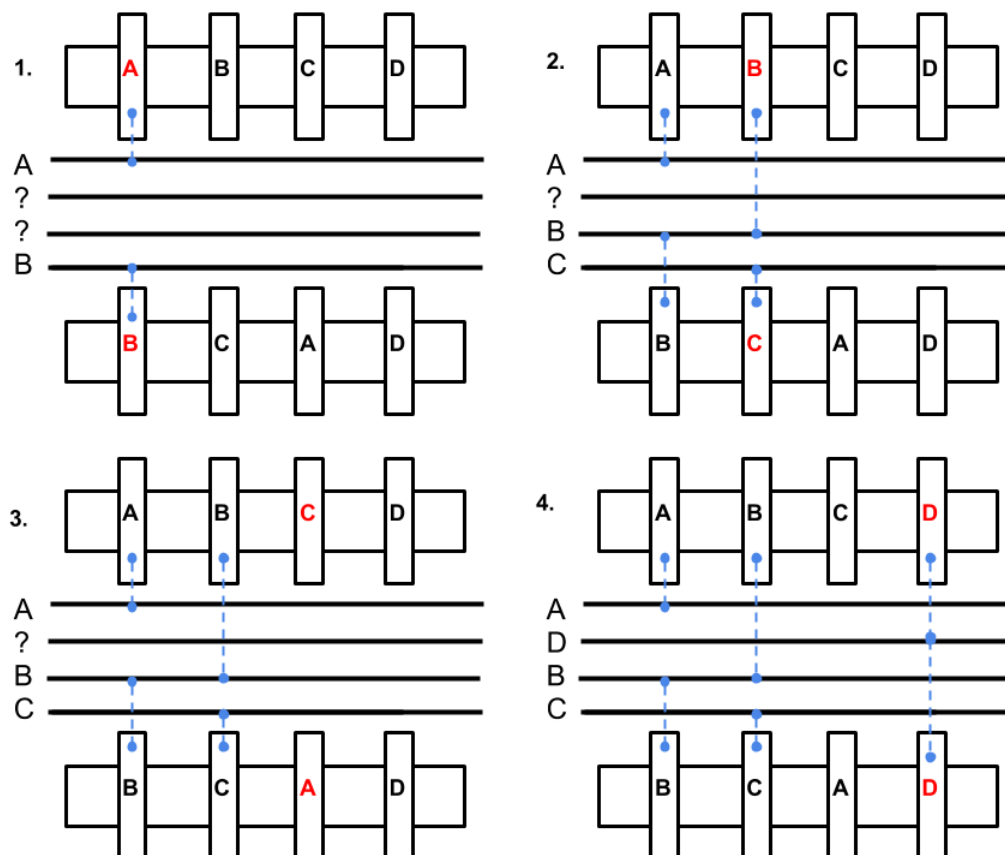


Figure 19: Diagram showing the steps of the algorithm that orders the routing channels to produce the smallest number of shorts.

routes are created, checks are added to look for intersections and space violations. Intersections are relatively straightforward to find based on the bounding boxes and layer of the geometry. To find space violations, it is necessary to know the design rule for the particular situation. This is done by creating dummy copies of the geometry being created and using **fgPlace** to place the bar. **fgPlace** will automatically place the geometry according to design rules. By measuring the distance between the placed dummy rectangles, it is possible to find the spacing constraint. If the new vertical bar violates these constraints, then a space violation has occurred.

In the case of either a space violation or intersection, a new bar is created and placed adjacent to the PDN. This new bar is renamed “X_split” and pins in the PDN that originally connected to gate “X” are renamed “X_split”. At this point, all of the existing vertical nets are destroyed and the routing is re-created with this new bar. (Due to a pointer error, the original version of this code actually renamed pins on both the PUN and PDN to “X_split”. This has since been fixed, but as will be described in the next section, in some cases, this error actually created more efficient

layout.) An example of how an intersection and space violation are corrected by adding another bar are shown in figures 20 and 21.

5.4.3 Removing Unused Routing Channels

As mentioned above, in an early version of the code, a pointer error resulted in both pins in the PUN and PDN being reassigned to the new routing channel. If a space violation or intersection is unavoidable with the original number of routing channels, this is a serious error that will result in an infinite loop as every additional routing channel that is created just results in more intersections or space violations. However, if it is possible to get DRC clean layout with the original number of bars, this error can be beneficial. In the event that this happens, there will be routing channels that have no vertical bars attached to them. Therefore, the code has a check to remove bars that are unused (leaving an empty routing channel). Figure 22 shows layout before and after unused bars are removed. For this particular layout, adding and removing bars creates a new ordering of the original routing channels that does not cause intersections.

For some solutions, it might be possible to remove this space and use the new order of the bars to create DRC clean layout. In other words, this loop could be used as a technique for finding the optimal order of the routing channels using information about the geometry. However, this typically does not work and is an extremely computationally expensive solution to the problem.

5.5 Contribution

The routing algorithms presented here are not particularly new. Routing is an extremely well studied problem and even papers published in the 1970s use more sophisticated algorithms than those developed here. The reason the routing routines in the BAG code base are interesting is they very much mirror the thought process that a circuit designer would use to layout geometry. For example, if a designer were going to hand draw layout in standard cell style, she would probably start with the channels in some (maybe arbitrary) order and then make vertical connections until she discovered an intersection. At that point, she would split the routing channels and rework the connections until there were no more intersections. Although this algorithm is somewhat less efficient in time and space than state-of-the-art routers, it does not require a computer science degree to understand. As stated before, one of the key problems in getting circuit designers to adopt automatic AMS generation is designers wanting to maintain control of the “art” of layout. Creating routing algorithms which mirror designers’ current workflow may give them more confidence about using AMS generators. Also it is worthwhile pointing out that with the move to smaller technology nodes that require grid-style layout, this method of layout may become common even in analog design. Finally, because this part of BAG is dealing with very low-level cells, sophisticated routers are not really necessary. The flexible framework of BAG also allows designers to integrate more complex routers, which is done to allow designers to retain control of the layout.

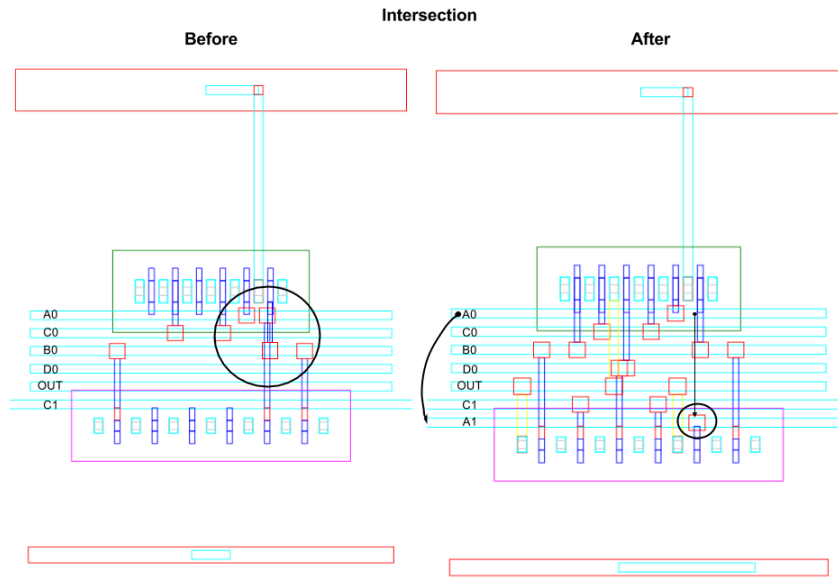


Figure 20: Layout on left shows an intersection. Layout on right shows intersection being removed by adding an extra routing channel.

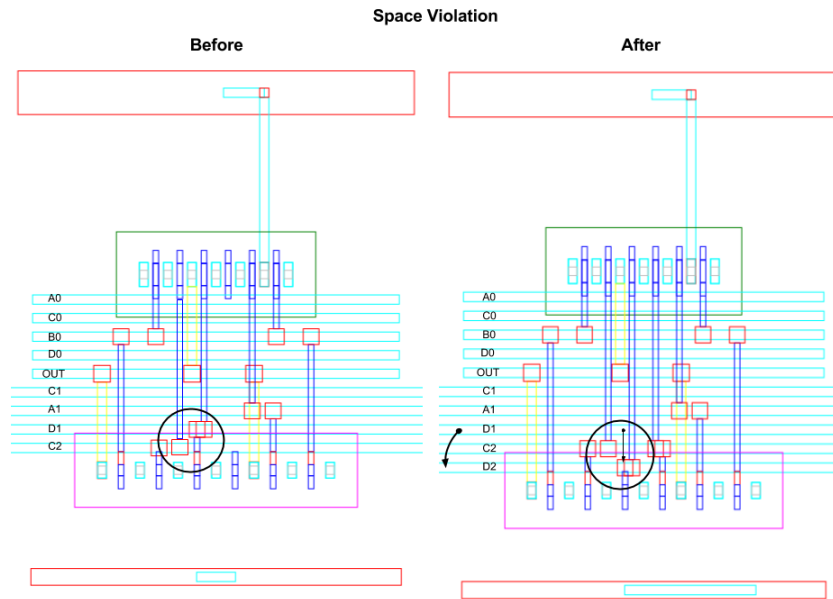


Figure 21: Layout on left shows a space violation. Layout on right shows the violation being corrected by adding an extra routing channel.

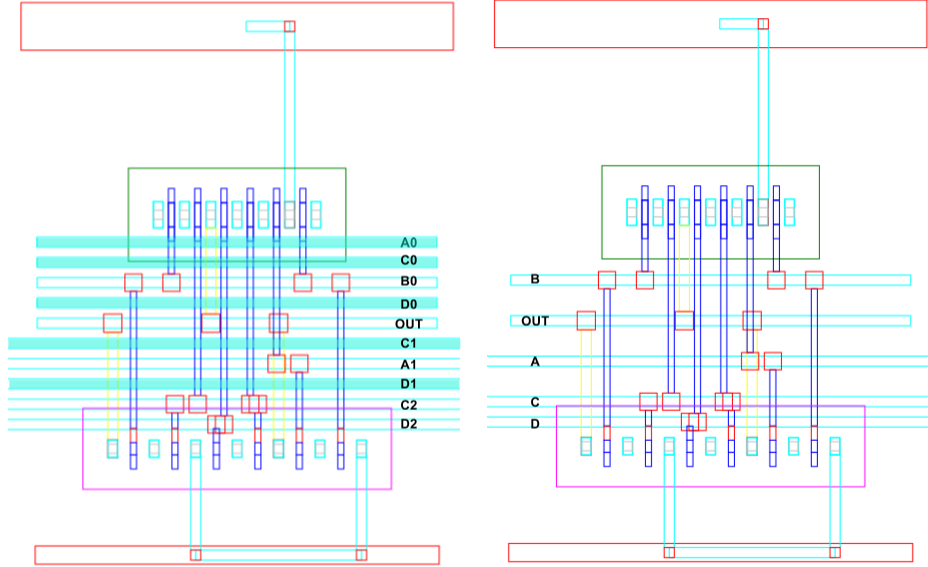


Figure 22: Layout before and after unused bars are removed.

6 Automatic Standard Cell Generation

6.1 Motivation

As discussed in the introduction, there have been decades of research developing analog layout tools that take a design from specifications to layout. In particular, the AMGIE synthesis environment is one of the most cited tools for carrying analog synthesis from design requirements to layout [7]. Although tools like AMGIE are very powerful, almost all existing analog synthesizers require a library of designs that are modified to meet certain specifications. As mentioned above, the goal of BAG is to create the generators in these libraries, which is one of the missing links in analog synthesis. BAG's goal is to simplify the generator designer's job. As such, BAG currently has tools to create sized schematics and to automatically generate layout given a designer-created template. Following the BAG philosophy of giving designers control of the generator design flow, designers should be able to choose the layout style (i.e. standard cell, standard block, amplifier, etc.) that best fits their design. However, once a style is chosen, a designer should not have to spend time manually filling in the template. Therefore, this work creates a new design class that automatically creates a layout templates once the layout style has been specified. Right now this has only been implemented for standard cells, but the framework could be easily extended to other layout styles.

This work began by automating custom digital standard cells because they are the most straight

forward. As early as the 1980s, CAD software was developed to place standard cells [21]. One of the breakthroughs for efficient custom digital layout came in 1981 when Uehara et al. published their foundational paper on using Euler paths to optimize the layout of complex boolean functions [22]. The following sections will describe how this technique is implemented in the BAG code base to automatically generate the templates for custom digital layout.

6.2 Creating Layout Without Diffusion Breaks

As mentioned, the current layout implementation requires the designer to fill in a **layout.info** template. To make this template more straightforward, it gives the designer the option of selecting series, parallel, shared source, shared drain, or single transistor layout. If the designer is implementing a more complex design, she has to create multiple device groups with diffusion breaks that must be connected. This is shown in Fig. 23. The circuit is laid out using the current routine as shown on the top right. Both the figure and the code snippet below (Current Implementation) show the parallel part of the network being separated from the series part. In the proposed implementation (shown in the second code snippet and the bottom right of Fig. 23), layout without diffusion breaks is produced. To produce this, it must be possible to create lists of the order of the gates and the order of the source/drain nets between the gates. With these lists, the terminals of the device group are set directly. The advantage of this is it is more space efficient. The disadvantage is it is more work for a designer to open a schematic and figure out the precise net names in order to make the nodelist. Therefore, this problem is optimally suited for automation.

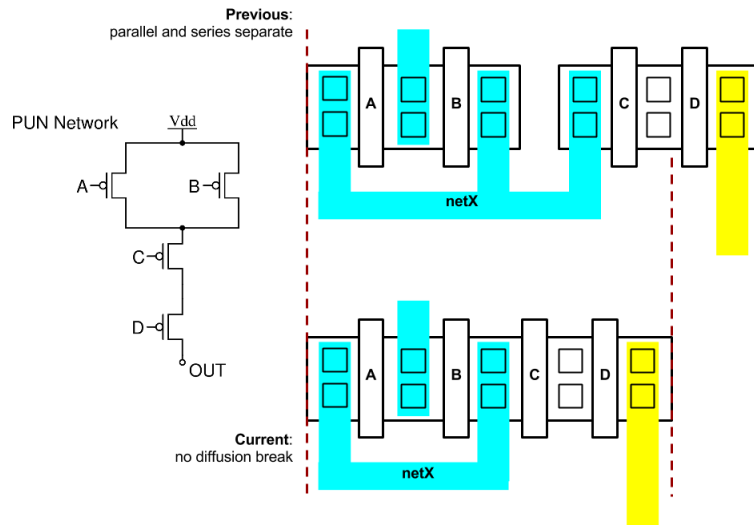


Figure 23: Layout of the boolean network on the left in the current implementation (top) and proposed diffusion-breakless implementation (bottom).

```

1 Layout_Info Snippet: Current Implementation
2
3 'name' : 'p_row1',
4 'type' : 'device_group',
5 'tran_type': 'pmos',
6 'height_ratio' : None,
7 'vertical_align' : "SOUTH",
8 'subelements' : [ { 'name': 'pmosPar',
9                     'param_mapping': {'W': 'Wp', 'L': 'L' },
10                    'G' : ['A', 'B'],
11                    'D' : ['netX'],
12                    'S' : ['VDD'],
13                    'B' : ['VDD'],
14                    'layout_type' : 'parallel',
15                },
16                { 'name': 'pmosSer',
17                  'param_mapping': {'W': 'Wp', 'L': 'L' },
18                  'G' : ['C', 'D'],
19                  'D' : ['netX'],
20                  'S' : ['OUT'],
21                  'B' : ['VDD'],
22                  'layout_type' : 'series',
23                }
24            ]
25
26 Layout\_Info Snippet: Proposed Implementation
27
28 'name': 'p_row1',
29 'type': 'device_group',
30 'tran_type': 'pmos',
31 'height_ratio': None,
32 'vertical_align': 'SOUTH',
33 'subelements': [ { 'name': 'pMOS',
34                    'param_mapping': {'W': 'Wp', 'L': 'L' },
35                    'G': ['A', 'B', 'C', 'D'],
36                    'nodeList': ['netX', 'VDD', 'netX', 'netY', 'OUT'],
37                    'B': ['vdd'],
38                    'layout_type': 'fully_custom',
39                }
40            ]

```

6.3 Finding Euler Paths

In order to automate the process of taking a schematic and producing the order of gates and source/drain nets to create layout without diffusion breaks, an old concept from graph theory is used. In 1736, Leonhard Euler solved the Seven Bridges of Königsberg problem by proving that a graph can only have a path that transverses every edge exactly once if the number of odd nodes is less than two. In his honor, such a graph is said to have an Euler path. In this work, the qualification that a graph with an Euler path must have fewer than two odd nodes is called the Euler criteria. A century and a half later, Fleury published a ground breaking paper in which he formulated an elegant algorithm for finding an Euler path in any graph that met Euler's condition [23]. Euler paths received new attention in 1981 when Uehara published a paper proving that CMOS layout can be created without diffusion breaks if an Euler Path can be found in the PUN and PDN [24].

6.3.1 Algorithms

In order to leverage Uehara's discovery to create compact layout, it is useful to understand the algorithm for finding Euler paths. A pseudo code version of Fleury's algorithm can be stated:

```
# edge = (start_node, edge, end_node)
# graph = [ edge1, edge2, edge3, ... ] where (1, A, 2) and (2, A, 1) are distinct

for edge in graph:
    euler_path = findEulerPath(current_edge, graph)
    if euler_path:
        break

# the returned euler_path contains both the Euler Path and the Node List
# note that the first start_node must be added to the Node List

def findEulerPath(current_edge, graph):
    remaining_graph = edges in that graph != current_edge
    connecting_graph = edges in remaining_graph where start_node = end_node of current_edge

    if len(remaining_graph) == 0:
        return [(current_node, end_node)]
    else:
        for edge in connecting_graph:
            path = findEulerPath(edge, remaining_graph)
            if path:
                return [(current_node, end_node)] + path
```

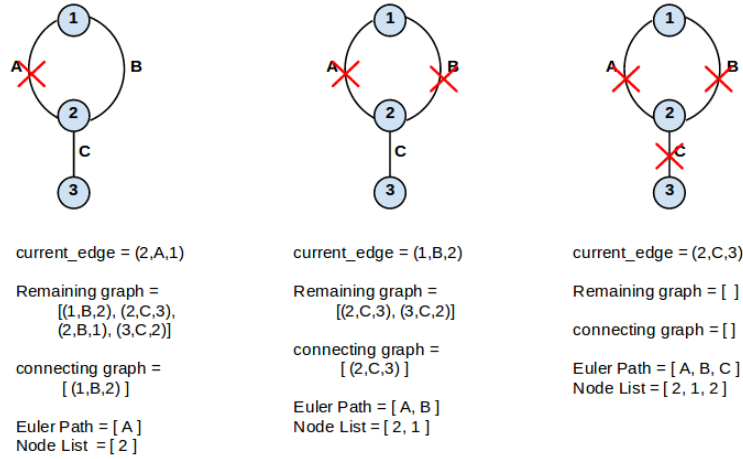


Figure 24: This figure shows the removal of edges from a graph as an Euler path is being built. Variables related to the pseudo code are also displayed.

The idea behind this algorithm is to create a list of connecting edges that contains all the edges in the graph. Starting with the first edge in the graph, remove this edge from the graph and look for connecting edges in the remaining graph. For each of these connecting edges, recursively call the function on that connecting edge and the remaining graph. As soon as all of the edges have been removed from the graph, return a list that contains the last edge that was traversed and the final node that was reached. Once this return happens, the algorithm begins returning up the stack of recursion. At each step in the recursion, the previous edge and node is added to the growing path and the new path is returned. This happens until the path reaches the top of the stack and is returned. This returned path contains both an Euler Path and a Node List (although the Node List has to be modified to add the starting node).

In the event that no Euler path is found, the for-loop at the top of the code just moves on to the next edge in the graph and tries the findEulerPath algorithm again. As soon as a path is found, this loop breaks. This is significant because this algorithm is able to find an Euler path, but there is always more than one (at the very least: ABC and CBA). Provided that the original graph meets the Euler specification, a path will always be found before the for-loop ends.

6.3.2 Converting a Circuit to a Graph

Taking a circuit schematic and turning it into a graph for Euler path finding is relatively simple. Each diffusion net of the circuit becomes a node in the graph and each gate becomes an edge between its source and drain nodes. This is illustrated in Fig. 25.

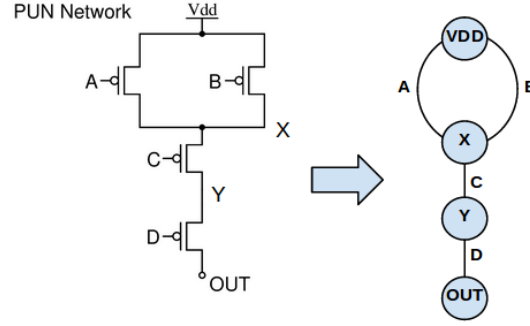


Figure 25: Conversion of a circuit schematic to a graph for Euler path identification.

6.3.3 Case 1: Consistent Euler Path

After converting the schematic to a graph, there are several possible things that could happen during the Euler path finding process. First, it might be possible that the same Euler path exists in both the PUN and PDN. If a consistent Euler path exists in both networks, it is advantageous to use it because it means PUN and PDN gates connecting to the same routing channels are closer to each other, which can reduce the length of routing channels.

A first pass glance at Fluery's algorithm suggests that to find the same path in the PUN and PDN, the for-loop at the top should be modified so every time a path is found it is added to a list:

```
# edge = (start_node, edge, end_node)
# graph = [ edge1, edge2, edge3, ... ] where (1, A, 2) and (2, A, 1) are distinct

euler_paths_PUN = []
for edge in PUNgraph:
    euler_paths_PUN.append(findEulerPath(current_edge, graph))

euler_paths_PDN = []
for edge in PDNgraph:
    euler_paths_PDN.append(findEulerPath(current_edge, graph))

compare euler_paths_PUN and euler_paths_PDN to find the common path
```

The problem with this modification is that it does not always work because the findEulerPath algorithm just returns *one* possible Euler path that begins with the **edge** in the graph. Since the findEulerPath algorithm just returns the first path it finds, the PUN and PDN may have a common Euler path that begins with the same edge that is missed. For example, in a three input Nand where the inputs are labeled **A**, **B**, **C**, the PDN findEulerPath algorithm might return **ABC**. Although that is a legitimate path in the PUN, it might not be found if the first path the algorithm

finds starting with edge **A** is **ACB**. Therefore, a better solution is to constrain Fleury's algorithm so it considers both the PUN and PDN graphs simultaneously and does not return until *both* the PUN and PDN graphs are empty. This solution was successfully implemented and Fig. 26 shows an example of a circuit and layout in which a common Euler path has been found in the PUN and PDN.

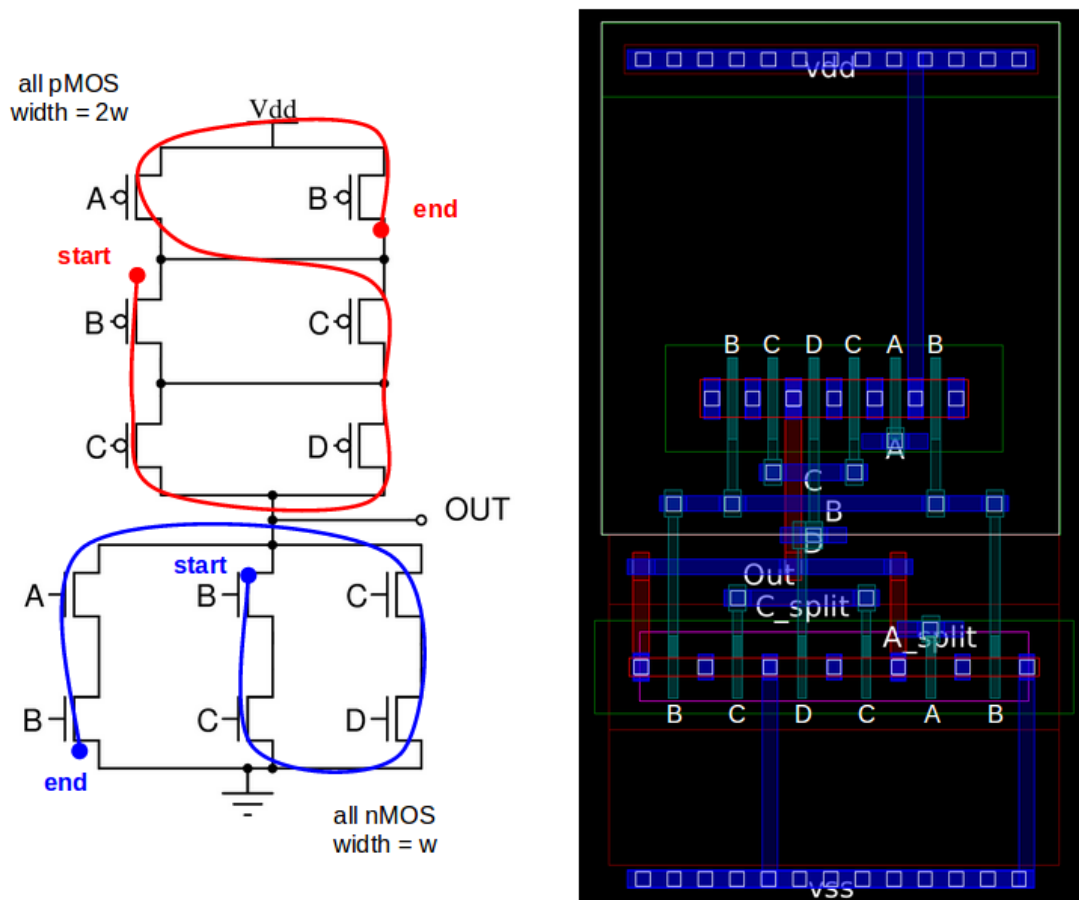


Figure 26: Automatically generated layout with the same Euler path in the PUN and PDN. The schematic on the left shows the Euler path found by the modified Fleury's algorithm.

6.3.4 Case 2: Different Euler Path in PUN and PDN

Another possible outcome is that the PUN and PDN both have Euler paths, but they are different. In this case, both networks are laid out without diffusion breaks, but the order of their gates is different. This results in slightly more parasitic capacitance, but is still space efficient.

6.3.5 Case 3: No Euler Path in either the PUN or PDN

It is possible either (or both) the PUN or PDN do not have an Euler path. This can happen if the network does not satisfy the Euler criteria or if the network has transistors of different widths that are not multiples of each other. The constraint on having the same width is due to the fact that a contiguous device group must be a uniform width.

Currently, not being able to find an Euler path is handled by modifying Fleury's algorithm to find the longest contiguous path in the graph and then returning the set of gates not included in the path. The process is then applied to the remaining gates to find the second longest path. This is continued until no gates remain. Each path becomes a separate device group in the network. This is shown in Fig. 27. Here the PUN does not meet the Euler criteria and the PDN has gates of different widths so both networks have multiple device groups.

6.3.6 Case 4: Euler Path in Devices of Different Widths

The previous case mentioned that devices of different widths had to be placed in different diffusion groups. In practice, if the widths have a common factor, it is possible to have devices of different widths in the same group if wider devices have more fingers. This is implemented by checking whether device widths have a common factor with other devices in the group. If so, a larger device is split into two (or more) devices in parallel (as shown in Fig. 28). A new Euler path is then found.

6.4 Mechanics of Moving from Schematic to Layout

Using Euler paths to map a schematic to a layout and having routing routines to automatically detect and remove shorts were the critical algorithmic pieces to automatically creating custom digital standard cells. However, in order to truly automate this process, there needs to be a way to transfer schematic and parameter information from the schematic sizing routines to the layout routines. This is done in `create_digital_layout` which is a method in the `PyNetlist` class. Specifically, after a BAG project is created, the `create_digital_layout` can be called on the `PyNetlist` object that is attached to the project. This method reformats information about the netlist and parameters and saves it to a file.

In addition, it creates a custom standard cell class for that cell. The custom standard cell class inherits from the standard cell class. It contains a subset of the methods in a regular standard cell class such as the `Nand` class shown in section 3.2.2. Specifically, a custom standard cell class has a dictionary of size parameters. This dictionary is automatically generated and contains the length, width, mult, and stack of each transistor in the layout. Both the default parameters and `layout_info` fields are empty because they are filled in when the cell is compiled. The custom class also has a `defineParamSpecs` method that defines a parameter pointing to the file where the netlist information is and another file that contains default parameter value information. An example of a custom standard cell class for an inverter is shown below.

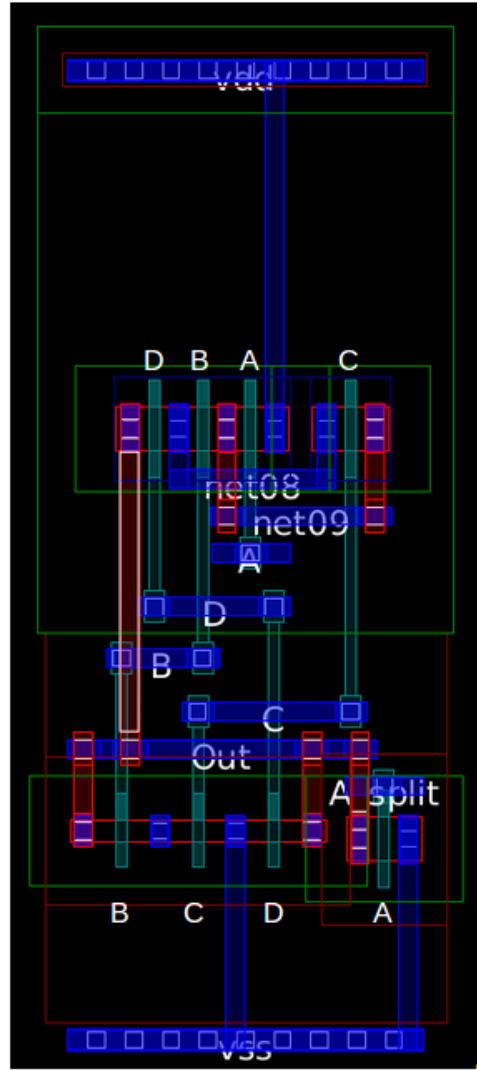
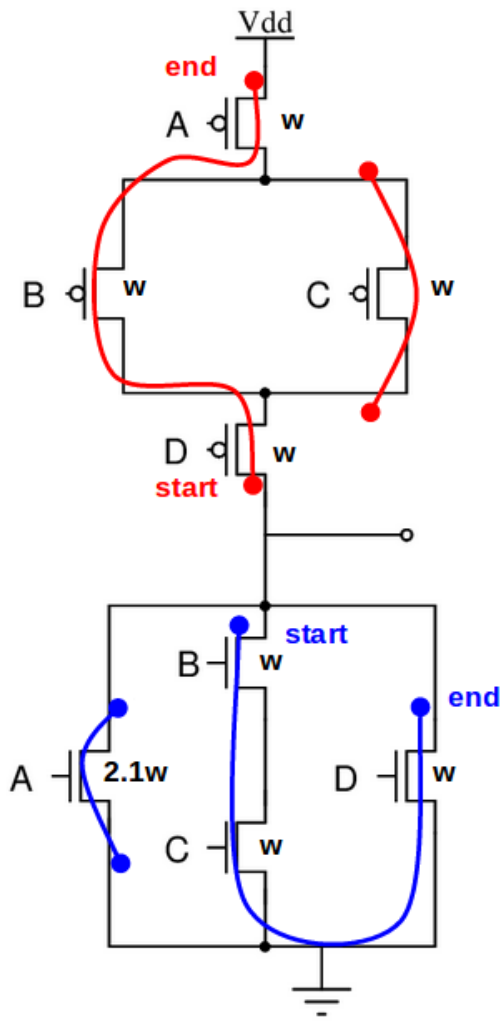


Figure 27: Automatically generated layout with no complete Euler paths. The PUN does not meet the Euler condition while the PDN has gates of different sizes.

```

1 ass inverter_custom( custom_std_cell ) :
2
3   paramNames = dict(
4     NMOmult = 'NMOmult',
5     NMOW = 'NMOW',
6     NMOL = 'NMOL',
7     NMOstack = 'NMOstack',

```

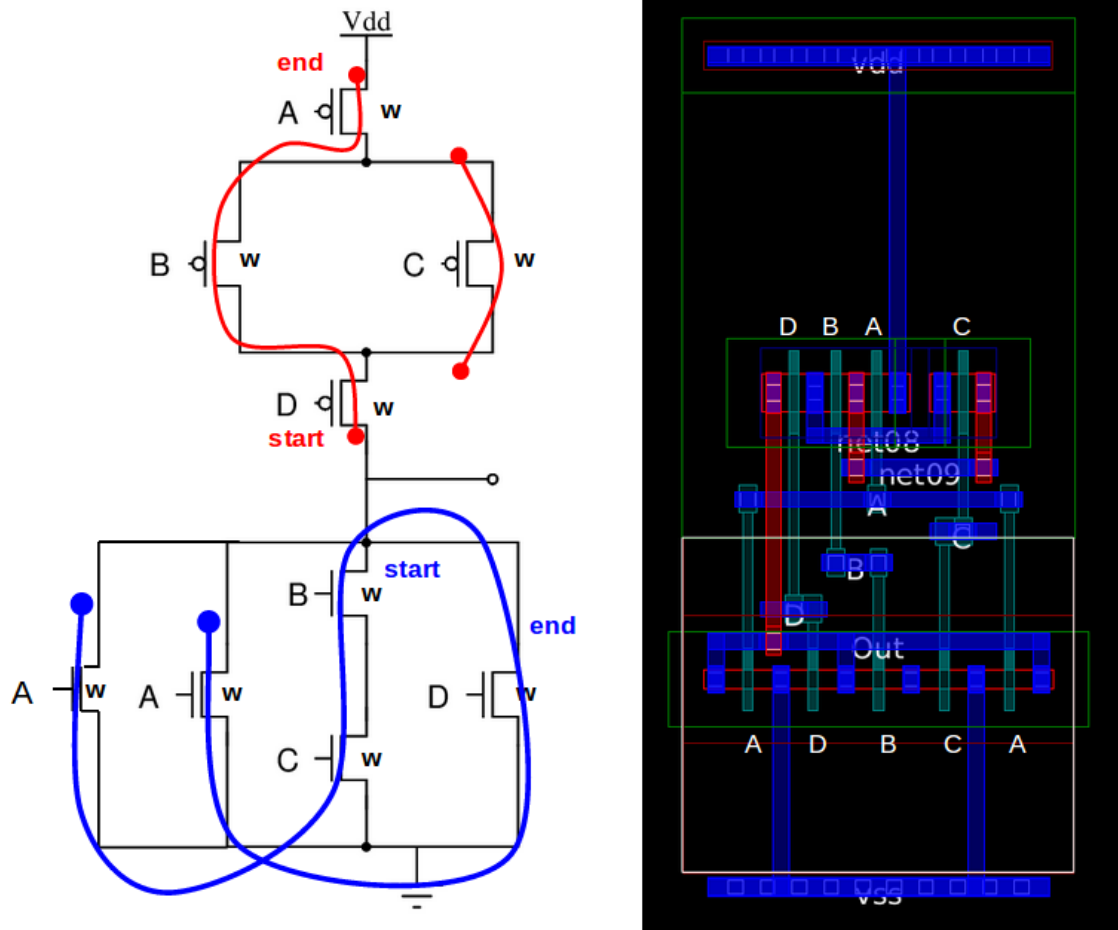


Figure 28: Automatically generated layout with a diffusion break on the top and a complete Euler path on the bottom because the bottom device has been split into two devices of the same width.

```

8     MOmult = 'MOmult',
9     MOW = 'MOW',
10    MOL = 'MOL',
11    MOstack = 'MOstack',
12 )
13
14 # Layout info is set dynamically, so initialize it for now.
15 layout_info = []
16 default = {}
17
18 @classmethod

```

```

19  @debugwrapper
20  def defineParamSpecs(cls, specs):
21      """Define the PyCell parameters. The order of invocation of
22      specs() becomes the order on the form.
23
24      Arguments:
25      specs - (ParamSpecArray) PyCell parameters
26      """
27
28      specs.add("file", ".../PDK/pycell_work/netlistInfo/inverter_custom.txt")
29      specs.add("parmFile", ".../PDK/pycell_work/netlistInfo/inverter_custom_paramspecs")
30      specs.add("useFile", True)
31
32      super(inverter_custom, cls).defineParamSpecs(specs)
33      mySpecs = ParamSpecArray()
34      # Parameter renaming: adds mySpecs (what you've defined above) into specs
35      renameParams( mySpecs, specs, cls.paramNames)

```

When a custom standard cell is compiled, the **defineParamSpecs** method gets overridden and all the parameters from the “paramFile” are added to the paramArray for that cell. Additionally, the saved netlist information from the “file” is processed using the Euler path algorithms and the algorithm for determining the order of routing channels. This information is then used to populate the **layout.info** dictionary. Once all the parameters and layout information have been added, the **defineParamSpecs** and **setupParams** methods are complete and **genTopology** is called by the Ciranova compiler. This method allows a custom standard cell to be generated automatically from files saved by the schematic sizing methods. However, because the layout information directory is populated while the cell is being compiled, it is more difficult to make manual modifications to the cell (for example, changing the order of routing channels). Therefore, when a custom standard cell is compiled, there is an option to also generate a filled in standard cell layout class that is printed to a file and can be manually modified and compiled.

6.5 Contribution

The goal of the BAG project is to create the library cells mentioned in papers such as [7]. In other words, BAG seeks to create analog generators that provide sizing, topology, and layout information with as much generality as possible. Current “library cells” require a large amount of sophisticated design and it is typically difficult to create new cells. BAG seeks to take the black magic out of generator creation. Although BAG’s sizing and layout routines are a step in this direction, the fact that layout classes must still be created by hand by the designer is a major drawback. Therefore, this work uses several algorithms to process schematic and layout style information provided by the designer to create fully automatic layout classes. Although this work has only produced automation for custom digital standard cells, it sets the stage for this to be extended to more generic standard

cells and even other layout styles. However, even having a few examples of cells that can be fully automatically generated provides motivation for BAG’s use as a “library cell” creator for powerful analog layout tools such as AMGIE.

7 Conclusion and Future Work

To summarize, the goals of this thesis were (1) to motivate analog synthesis by showing some examples of fully automatic generators and (2) to create a code framework for fully automatic standard cells that could be extended to other styles of layout. The sections above approached these goals in the following way:

- Section 3 described the complete BAG code base to show the ease with which a designer can create a circuit in BAG. This section can also serve as documentation to future BAG users.
- Section 4 described the debugger tool that can aid designers in understanding how the code actually builds their layout. This debugger can also help them figure out how to make minor modifications to automatically generated cells.
- Section 5 discussed the improvements to the routing algorithms to prevent shorts. These changes were instrumental in automatically creating layout templates that did not require designers to manually modify the code to remove shorts.
- Section 6 talked about the custom standard cell class. This class contains algorithms to determine how to place devices and order routing channels based on information extracted from the netlist. This section sets up the basis for extending fully automatic generator creation to other layout styles.

7.1 Future Work

There are several additional steps that could be taken to expand this work:

- The next major step in this work would be extending the fully automatic custom digital standard cells to other layout styles. To begin with, automatically generating layout templates could be extended to all standard cells by making it possible to have multiple PUN or PDN groups.
- Also, the amplifier class needs to be written. Once this is done, code could be developed to automatically create layout templates for this class.
- Across the BAG codebase, routing is currently done in different ways depending on the style of layout. The routing algorithms in standard cell should be rewritten to use the **std.route** class to make routing consistent across the codebase.
- Currently, when routes are split, the new nets are not connected. Code should be written to connect these routing channels without creating shorts.

A BAG Sizing Routines

1. BAG_project
 - (a) The BAG Project is where all the information for a particular BAG session is stored.
 - (b) This includes things like the global session parameters and a hook to the virtuoso session so it does not have to be opened and closed repeatably.
 - (c) It also launches Skill scripts that do parsing of circuit properties such as CDF parameters.
2. DesignDescription Module
 - (a) Module
 - i. A module defines a circuit and all its metadata including parameters and testbenches.
 - ii. The module has links to objects related to the circuit such as testbenches and parameters.
 - iii. It also has fields with data structures representing nets, pins, connections, and names.
 - iv. Information about the cell's parent as well as child instances is stored in the object.
 - v. Most of its methods relate to getting and setting fields.
 - (b) Design Module
 - i. The design module is one of three main child class of module that has methods to do initialization of a single cell.
 - ii. Specifically, the methods instantiate any subcircuits and testbenches.
 - (c) Mos Module, Cap Module, Res Module
 - i. These are children classes of Design Module and represent BAG primitives.
 - ii. The only additional functionality they add are sizing methods (**size_for**).
 - (d) Testbench Module
 - i. The design module is the second main child class of module.
 - ii. Contains methods which instantiate subcircuits within a testbench including a DUT. Once it instantiates the DUT, it replaces this with a pointer to the DUT that called it - see Fig. 5 for an explanation.
 - iii. It contains information related to simulation (initial conditions, etc.) as well as a simulator interface.
 - iv. Has the ability to launch simulations.
 - v. Adds probes (measure class), changes testbench parameters and initial conditions.
 - (e) VarStruct Module
 - i. This is the third main child class of module.
 - ii. It is a module whose circuit structure is dependent on a parameter.

- iii. It contains a pointer to the original instance.
 - iv. It also contains a structural parameter with a callback - when changed, it regenerates the structure.
 - (f) VarStructDesign Module
 - i. Creates module that inherits from VarStructModule and DesignModule.
 - (g) VarStructTestbench Module
 - i. Inherits from VarStruct Module and Testbench Module.
 - ii. Checks to see if a flag is raised to restructure. If so, it recreates the structure before running.
 - iii. Modifies the object structure module (pins, nets, connections, instances, parent, etc.) and creates the netlist when it's needed.
 - (h) ArrayModule and TreeModule
 - i. Inherit from VarStruct Modules.
 - ii. Represents structure as an array and tree, respectively.
 - (i) PyNetlist
 - i. Contains a list of Netlist cells (subckts) and a pointer to the top one.
 - ii. Has functionality for modifying the netlist (unlike a static string representation).
 - (j) Data
 - i. Abstract data class representing some piece of circuit data with units.
 - ii. Can set units and bounds.
 - iii. Used by measures in parameters and testbench.
 - (k) Measure
 - i. Used for probes in testbenches.
 - (l) Corners
 - i. Used in testbench for simulation.
 - ii. Uses technology class to find PDK information about corners.
 - (m) Parameters
 - (n) Design Parameters
 - i. Object of module.
 - ii. Adds pointers to parameters at different levels : eg. parameter in mos (nW) points to parameter at amp level (W).
3. Interface Module
- (a) Simulator Interface
 - i. Contains an interface for running various simulator scripts.
 - (b) Ocean Simulator

- i. Handles parsing of the ocean script.
 - ii. Takes the parameterized ocean script created in **import_schematic_from_template** and uses it as a template that is filled in with sizes from the pyNetlist.
 - iii. Also tells virtuoso to run script.
 - (c) Spectre Simulator
 - i. Only used to generate lookup tables used in **size_for** function.
 - ii. This is currently under development.
 - (d) OA Interface
 - i. Interface to open access database (database that stores symbol, schematic information).
 - ii. This takes the abstract BAG Project and turns it into a “BAG View” and a “yaml view” that can be stored in a Cadence library and viewed by Cadence. The goal is that if Cadence libraries were copied, the recipient could view sized BAG schematics.
 - iii. Specifically it is used to create a sized schematic by copying a template schematic and writing in the actual device sizes and transistor process information. This is done so (1) The schematic can be viewed for confirmation (2) LVS can be run and (3) so tests can be run manually.
 - iv. Also used in BAG project during **import_schematic_to_template** to create yaml files, read the yaml files, and to create a BAG class.
 - (e) Skill CDF Interface
 - i. Runs specific skill scripts.
 - ii. Writes a script and runs it in Virtuoso (doesn’t create a separate file, just runs the commands).
 - iii. Also reads/writes CDF parameters using skill API.
 - (f) Technology
 - i. Reads in BAGtech.yaml (the BAG format of the PDK).
 - ii. This is added as an object of BAG project.
 - iii. Used by OceanSimulator to set the corners of transistors and details about primitives.
- #### 4. Workbench Module
- (a) Shell
 - i. Wrapper for Pexpect.
 - ii. Pexpect is the module for running Virtuoso - it launches commands to the OS at specified times.
 - (b) DB Access
 - i. Implementation of shell to run Cadence programs (Skill scripts, DB access, or Virtuoso).

B BAG Layout Routines

1. BAG PyCell

- (a) This is the parent for all BAG PyCells.
- (b) It overrides the Ciranova DloGen's (dynamic layout object generators) methods (DefineParameterSpecs and setupParams) with some BAG specific parameters.
- (c) It also includes a few other methods that are used for setting up pycells.

2. BAG Primitive PyCell

- (a) Overloads defineParmSpecs and SetupParameters for primitive devices (resistor, transistor, etc.).

3. BAG Primitive Transistor

- (a) Contains methods to setup dummies and add pins to transistors.

4. Routing Methods

- (a) Contains methods to create routing in standard cells.
- (b) Create Routing Channel: creates horizontal routing channels for each net between device groups.
- (c) Straight Line Instpin to Bar: draws vertical bars between instance pins and routing bars.
- (d) Restretch Routing Channels: Stretches out routing bars after they have been created so they span the length of the standard cell.
- (e) Also contains methods for calculating sizes (so routes are placed without violating design rules) and adding pins/contacts.

5. Std Cell

- (a) Overloads the **genTopology** and **genLayout** methods to create netlist and layout information.
- (b) Has methods that determine the number of fingers transistors should have to fit within the height constraints.
- (c) Has methods to place and connect power rails.
- (d) Calls methods from Routing Methods to connect devices to routing bars.
- (e) Does placement of device groups, rails, and route groups.
- (f) This is the parent class for most layout formats.

6. Custom Std Cell

- (a) This method (in combination with some modifications to the sizing methods in BAG) is used to automatically create digital layout.

- (b) It uses saved information about the netlist to fill in the **layout.info** data structure.
- (c) It contains methods to find Euler paths to create transistor rows without diffusion breaks.

7. Std Cell Row

- (a) Inherits from std cell.
- (b) Creates a row of adjacent, interconnected standard cells.

8. Std Block

- (a) Inherits from std cell.
- (b) Creates a block of interconnected standard cells.

9. Std Route

- (a) Eventually this should replace Routing Methods. It contains many of the same methods but creates routing objects to make it easier to modify routing.

References

- [1] V. Grimblatt, "Synthesis - state of art," in *Devices, Circuits and Systems, Proceedings of the 6th International Caribbean Conference on*, 2006, pp. 327–332.
- [2] A. Sangiovanni-Vincentelli, "The tides of eda," *Design Test of Computers, IEEE*, vol. 20, no. 6, pp. 59–75, 2003.
- [3] R. Brayton and J. Cong, "Electronic design automation past, present, and future," in *NSF Workshop Report*, 2009.
- [4] R. Harjani, "Oasys: a framework for analog circuit synthesis," in *ASIC Seminar and Exhibit, 1989. Proceedings., Second Annual IEEE*, 1989, pp. P13–1/1–4.
- [5] J. R. Koza, I. Bennett, F.H., D. Andre, M. A. Keane, and F. Dunlap, "Automated synthesis of analog electrical circuits by means of genetic programming," *Evolutionary Computation, IEEE Transactions on*, vol. 1, no. 2, pp. 109–128, 1997.
- [6] R. Martins, N. Lourenço, and N. Horta, "Laygen ii: automatic analog ics layout generator based on a template approach," in *Proceedings of the fourteenth international conference on Genetic and evolutionary computation conference*, ser. GECCO '12. New York, NY, USA: ACM, 2012, pp. 1127–1134. [Online]. Available: <http://doi.acm.org/10.1145/2330163.2330319>
- [7] 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," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 20, no. 9, pp. 1037–1058, 2001.
- [8] P. A. e. a. Crossley, J., "A designer-oriented integrated framework for the development of ams circuit generators," 2013.
- [9] V. Aggarwals, "Analog circuit optimization using evolutionary algorithms and convex optimization," Master's thesis, Massachusetts Institute of Technology. Dept. of Electrical Engineering and Computer Science., May 2007.
- [10] K. Kumar, P. Prem; Duraiswamy, "An optimized device sizing of analog circuits using particle swarm optimization," *Journal of Computer Science*, vol. 8, no. 6, p. 930, 2012.
- [11] P. Maulik, L. Carley, and D. Allstot, "Sizing of cell-level analog circuits using constrained optimization techniques," *Solid-State Circuits, IEEE Journal of*, vol. 28, no. 3, pp. 233–241, 1993.
- [12] M. J. M. Pelgrom, A. C. J. Duinmaijer, and A. Welbers, "Matching properties of mos transistors," *Solid-State Circuits, IEEE Journal of*, vol. 24, no. 5, pp. 1433–1439, 1989.
- [13] K. Lakshmikumar, R. Hadaway, and M. Copeland, "Characterisation and modeling of mismatch in mos transistors for precision analog design," *Solid-State Circuits, IEEE Journal of*, vol. 21, no. 6, pp. 1057–1066, 1986.

- [14] L. Zhang, N. Jangkrajarn, S. Bhattacharya, and C. J. R. Shi, "Parasitic-aware optimization and retargeting of analog layouts: A symbolic-template approach," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 27, no. 5, pp. 791–802, 2008.
- [15] F. Balasa, S. Maruvada, and K. Krishnamoorthy, "On the exploration of the solution space in analog placement with symmetry constraints," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 23, no. 2, pp. 177–191, 2004.
- [16] *Ciranova Python API Reference Manual*, Ciranova, Inc, April 2009.
- [17] M. A. Breuer, "A class of min-cut placement algorithms," in *Proceedings of the 14th Design Automation Conference*, ser. DAC '77. Piscataway, NJ, USA: IEEE Press, 1977, pp. 284–290. [Online]. Available: <http://dl.acm.org/citation.cfm?id=800262.809144>
- [18] R. L. Rivest and C. M. Fiduccia, "A greedy channel router," in *in Proc. 19th Design Automation Conf*, 1982, pp. 418–424.
- [19] T. Yoshimura and E. Kuh, "Efficient algorithms for channel routing," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 1, no. 1, pp. 25–35, 1982.
- [20] J. Cohn, D. Garrod, R. Rutenbar, and L. Carley, "Koan/anagram ii: new tools for device-level analog placement and routing," *Solid-State Circuits, IEEE Journal of*, vol. 26, no. 3, pp. 330–342, 1991.
- [21] A. Dunlop and B. W. Kernighan, "A procedure for placement of standard-cell vlsi circuits," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 4, no. 1, pp. 92–98, 1985.
- [22] T. Uehara and W. VanCleemput, "Optimal layout of cmos functional arrays," in *Design Automation, 1979. 16th Conference on*, 1979, pp. 287–289.
- [23] Fleury, "Deux problemes de geometrie de situation," *Journal de mathematiques elementaires*, p. 257–261, 1883.
- [24] M. Grotschel and Y. Yuan, "Euler, mei-ko kwan, konigsberg, and a chinese postman," in *Documenta Mathematica*, 2010, pp. 43–50.