Closing the Analog Design Loop with the Berkeley Analog Generator

Nicholas Werblun
Vladimir Stojanovic, Ed.

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2019-23
http://www2.eecs.berkeley.edu/Pubs/TechRpts/2019/EECS-2019-23.html

May 1, 2019

Closing the Analog Design Loop with the Berkeley Analog Generator

by

Nicholas Werblun

A thesis submitted in partial satisfaction of the

requirements for the degree of

Master of Science

in

Electrical Engineering and Computer Sciences

in the

Graduate Division

of the

University of California, Berkeley

Committee in charge:

Professor Vladimir Stojanović, Chair
Professor Borivoje Nikolic

Spring 2019

Closing the Analog Design Loop with the Berkeley Analog Generator

---

Closing the Analog Design Loop With BAG

by Nicholas Werblun

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:

Professor Vladimir Stojanovic
Research Advisor

4/12/19
(Date)

* * * * * * *

Professor Borivoje Nikolic
Second Reader

4/30/2019
(Date)

Abstract

Closing the Analog Design Loop with the Berkeley Analog Generator

by

Nicholas Werblun

Master of Science in Electrical Engineering and Computer Sciences

University of California, Berkeley

Professor Vladimir Stojanović, Chair

Analog and mixed signal IC design is notoriously difficult and slow due in large part to the layout. Modern integrated circuit fabrication with such small devices can have significant interconnect parasitics that can drastically affect the behavior of a circuit's design. The implication is that simulations of circuit's behavior are unreliable until after the interconnect parasitics are extracted from the layout and included in the simulation.

The Berkeley Analog Generator (BAG) is a Python-based tool that interfaces with the Cadence Virtuoso software [3] that aims to solve the above problem. BAG allows the user to write parametrizable generator scripts that will automatically generate the entire layout and schematic, as well as run the layout-versus-schematic (LVS) and post-layout extraction (PEX) tools and export the results in a time that ranges from seconds to minutes based on circuit complexity. Designers who have decided on a certain topology can write a layout and schematic generator script in a high level programming language with class based hierarchy once, and then any changes in the circuit simply require changing the corresponding parameters file containing the circuit specifications. Additionally, BAG allows the automation of simulation and post-processing of simulation data as well as implementation of higher-level design scripts that encapsulate designer insights and methodology, as well as opens the doors for automated optimizer-driven circuit design.

This report shows examples of many common circuit blocks and their BAG implementation in an advanced process node, as well as an example of how BAG can be used to speed up the design process. Although the generator scripting offers the implementation in a higher-level language, certain implementation strategies and methodologies work better than others, and this report aims at presenting a systematic generator writing methodology and illustrates it on a set of typical analog-mixed signal blocks found in a high-speed link front-end. In three months, a library of generators ranging from small basic circuits to entire receiver chains were written; then in roughly two weeks, an LVS/PEX verified design for a 25Gbps optical communication link receiver in a 14nm FinFET process was created using BAG cells and test benches. Further possibilities and uses of BAG are also discussed.

Contents

Contents **i**

List of Figures **iii**

List of Tables **iv**

Listings **iv**

1 What Makes Analog Design Difficult? **1**
1.1 Modern Mixed Signal IC Flow and Concerns 1

2 The Berkeley Analog Generator **4**
2.1 Overview and Top Level 4
2.2 Layout Generators 5
2.3 Schematic Generators 20
2.4 Design Manager and Test Benches 22

3 Example Generator Implementations **25**
3.1 Starting Small: Passive Loaded Differential Amplifier 25
3.2 Getting Fancier: A Double Tail Sense Amplifier and Latch 26
3.3 Endless Possibilities: An Entire Receiver Analog Front End Receiver or Two 28

4 Design Problem: An Optical Receiver **33**
4.1 Problem Statement and Architecture 33
4.2 Transimpedance Amplifier 35
4.3 Continuous Time Linear Equalizer 38
4.4 Preamplifiers 40
4.5 Comparator 40
4.6 Design Verification 44

5 Conclusions, Future Work and Other Possibilities **49**
5.1 How BAG Saved the Day 49
5.2 Future Work 50

5.3 Related Work and Other Possibilities 50

Bibliography **52**

List of Figures

2.1 BAG Overview 4
2.2 Differential amplifier schematic template example 6
2.3 Base layout 12
2.4 Transistor wiring 13
2.5 Finished layout (only metals shown) 14
2.6 Differential amplifier layout with various widths and number of finger choices. . 15
2.7 Differential TIA and CTLE 17
2.8 Various resistor DAC generated layouts 19
2.9 Various resistor DAC schematics 21

3.1 Differential amplifier layout 26
3.2 Passive diff-amp with resistor DAC 27
3.3 DTSA layout 28
3.4 DTSA with offset correction 28
3.5 DTSA with latch 29
3.6 Various generated AFE layouts 31
3.7 Second front end 32

4.1 System architecture 34
4.2 CTLE effect on a generic one-pole system 35
4.3 TIA Schematic 36
4.4 TIA low frequency small signal model 37
4.5 CTLE Schematic 38
4.6 CTLE Half-Circuit 38
4.7 Preamp Schematic 40
4.8 Double Tail Comparator Schematic 41
4.9 DTSA Typical Operation 42
4.10 DTSA Noise 43
4.11 AFE AC Performance 44
4.12 Front End Noise PSD 45
4.13 Sampler Input Eye Diagram 46
4.14 PRBS Pattern Response 47

List of Tables

4.1 Performance Summary 48

Listings

2.1 Creating rows for transistors 8
2.2 Assigning transistors to a row and initializing 9
2.3 Assigning transistors to columns 10
2.4 Determining transistor drain/source configurations 11
2.5 Drawing wire connections. 13
2.6 Adding and placing templates to a layout 16
2.7 Resistor DAC params sample 18
2.8 Resistor DAC schematic generator 20
2.9 Test bench parameters 23
3.1 Track manager setup 27
3.2 Track manager usage 27

Acknowledgments

Thank you to professor Vladimir Stojanović for the advising and guidance through this project. Starting is often the hardest part, as is finding a direction to follow. Huge thank you to Sidney Buchbinder and Ruocheng Wang for mentoring me through my degree. Thank you for answering my multi-hundreds of questions and teaching me roughly one semester's worth of information in just a few weeks total. I asked for design experience and I got it; and you helped me through it.

Additional thanks to Olivia for telling me to join the group. Thanks to Kourosh for making the test bench generators and explaining them to me. Finally a thanks to all the rest of team Vlada: Krishna, Panos, Christos, Pavan, Nandish, Taewhan and Eric for making my research experience go smoothly. Finally, thank you to Eric Chang et. al. for creating BAG so that I never had to do layout by hand.

Chapter 1

What Makes Analog Design Difficult?

If you know anyone who fits into the category of analog, RF or mixed signal designers, you may already be familiar with the disheartened, defeated look they often display. Is it that IC design attracts people of this nature? Or is it that IC design slowly gnaws at the existence of engineers until they are but a shell of their former selves? For this report, we will assume the latter. We will also assume this is due to standard practices more than any deep-rooted truths about IC design or people in general.

1.1 Modern Mixed Signal IC Flow and Concerns

IC design is slow. The fabrication process for an IC is very complex, and for complex chips the upfront cost can be immense [5]. Mistakes are costly both monetarily and time-wise, ranging from hundreds of thousands of dollars and months to millions of dollars for simple edits to a circuit. Furthermore, a company selling chips for use in other companies' products are liable for their chip's performance. Should their component be the reason for a product's failure, that company may then be responsible for reimbursing the cost of the product. For a $1000 cell phone with 100 million units shipped per year, a 1% chip failure rate can lead to costs in the billions. For a company designing ICs then, it is crucial to get a high yield of functioning chips on the first try to minimize costs.

Since a chip cannot be tested until after it is manufactured, IC designers tend to simulate the worst possible scenarios. Semiconductors behave quite differently under varying temperatures and circuit conditions may not be stable. Supply voltages may vary, manufacturing can create component offset that negatively impacts the circuit, static discharge events can occur and in modern processes, transistors age with time and exposure to bias voltages .Due to this, most companies will only tape out a chip when they are fully confident it will function properly under all use cases and over guareteed lifetime.

Ignoring the design procedure and assuming an architecture and component parameters are already chosen, the testing procedure is generally as follows:

1. Simulate the chosen design across all specs to ensure proper behavior.

2. Simulate again, but at extreme temperatures and supply changes.
3. Simulate again, but including random process variation.
4. Draw the layout and extract the parasitics.
5. Perform 1. through 3. on the extracted layout.

If the circuit fails at any of these steps, the design must be modified and the steps repeated. Only when all of tests are passed and verified through potentially hundreds of different tests will the chip be sent for fabrication. Depending on the complexity of the chip and the size of the team, this process can take anywhere from months to a year or more.

Why Does it Take So Long?

Layout.

Why Does Layout Take So Long?

Laying out a circuit is the term used to describe the process of how an abstract schematic is translated into a physical picture that then turns into mask layers that the foundry uses in the fabrication process to define where to dope the semiconductor, add connections and place and route metals to generate an accurate depiction of the desired device parameters and sizes. This means that the designer must manually draw active regions, metal wire sizes and routing, vias, gate connections, etc. There are an infinite number of possibilities to generate such a layout, but each comes with some trade off. Perhaps making a wire very long and routing it around a structure is much easier, but introduces significant IR drop. Perhaps a wire needs to carry more current through it, so the width is increased, but this introduces extra parasitic capacitance. These trade offs are some of the main concerns of both layout engineers and circuit designers.

Layout bottlenecks the design flow due to the slow turnaround time. With the reduction in device sizes over the years, the design rules imposed by the fabricators have become more strict and the parasitics in conjunction with high frequencies have started to have a more significant effect on the circuit's post-layout performance [1] [7]. In deeply-scaled processes, layout patterns also induce systematic pattern-dependent variations and difference in strain which affects transistor performance. Layout then, is a crucial part of the verification step. Since the process is slow and precise, this means that any changes require significant work to modify the layout and ensure nothing was incorrectly altered in the process.

The time frame of months to years is believable then if the turnaround for hand-drawn layout is roughly a couple days to a week for fairly insignificant design modifications like transistor resizing, or passive device resizing. Each and every time the designer receives the extracted layout and modifies the components to correct for changes, they must wait days

before they can start the cycle again. By Amdahl's law then [1] [2], speeding up the layout should allow one to close the design loop faster and reduce the pain of IC process.

[1] Amdahl's law is generally referenced in computer program runtime, but the concept of speeding up fractions of a process applies here as well.

Chapter 2

The Berkeley Analog Generator

2.1 Overview and Top Level

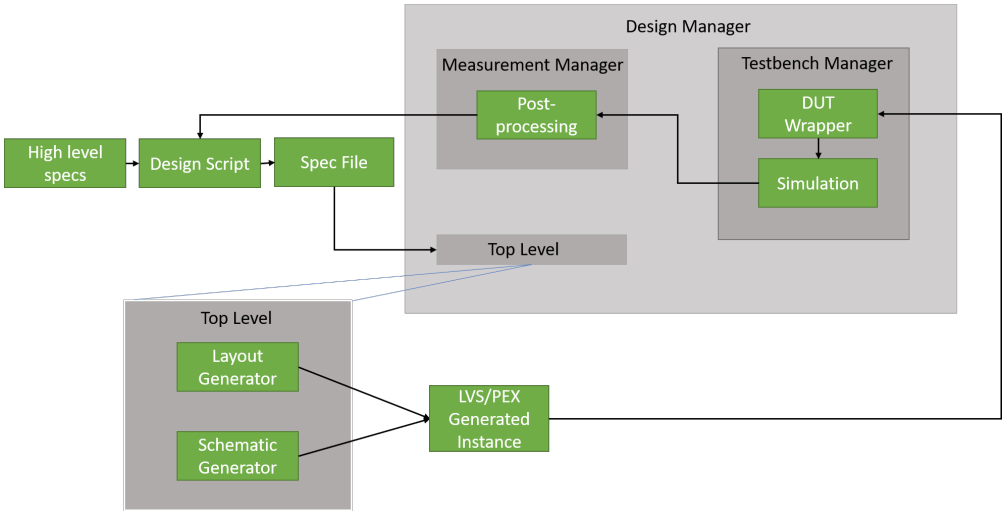


Figure 2.1: BAG Overview

As discussed in [3], BAG is a framework that allows users to create, use and test process-portable analog generators. Designers can create template schematics and write a scripted layout generator that incorporates their design methodology in a technology agnostic, parametrized way. At the highest level, the user inputs parameters, examples of which will be shown later in this chapter and in Chapter 3, such as device dimensions and passive component values into a specification file, and a script will generate a schematic, LVS tested layout and a PEX netlist. The main advantage is that the delay of the design loop discussed in Chapter 1 is significantly reduced as post-layout effects can be directly included into the flow.

As will be discussed in later sections, there are a few main components to a "complete" generator. At the lowest levels are the layout generator and schematic generator. These are responsible for the physical process of generating the circuit representation and layout. In order to use these, there is also a top level script that reads a parameter file and runs the schematic/layout generators with these specifications. The top level is also responsible for deciding whether or not to run LVS/PEX on the generated instance.

At an even higher level is the notion of a design script and design managers. Design manager is a class responsible for using the top level generator mentioned previously and overseeing the process of running tests and post-processing on test results. Design manager has associated test bench scripts which are responsible for connecting the generated device into a previously made test harness. The test bench script maps the pins of the instance to the pins of the test harness and runs predetermined SPICE simulations (i.e. AC, transient, S-parameters) before exporting the results to Python. Design manager can then pass the results to a measurement manager which can process, plot, etc. The entire process can be visualized as in Figure 2.1.

The notion of a design script is an even higher level concept which allows a designer to encode their design procedure automatically into a close-looped script. The user can write a script that computes passive and transistor sizings, allow design manager to generate and test the post-PEX netlist, and iterate based on the results. The possibility of incorporating a design script will be discussed later, although a design script is not presented in this work. Design scripts are discussed in further detail in [3] and [6].

2.2 Layout Generators

Layout generators are arguably the most important portion of the process and are extensively discussed in [3]. BAG offers a set of functions for drawing transistors, automatically routing metals, drawing vias, etc. to allow the same capability that hand design would produce. The goal of the designer is to use these functions in a generic way that automatically computes where and how to draw connections that will be DRC and LVS clean regardless of the input specifications.

The typical process of generating a large complex block is to start with small cells, like an inverter or an active diff amp using `AnalogBase`. `AnalogBase` is a class in BAG used for drawing layouts comprised entirely of transistors. Firstly, the user draws a schematic template, like shown in Figure 2.2.

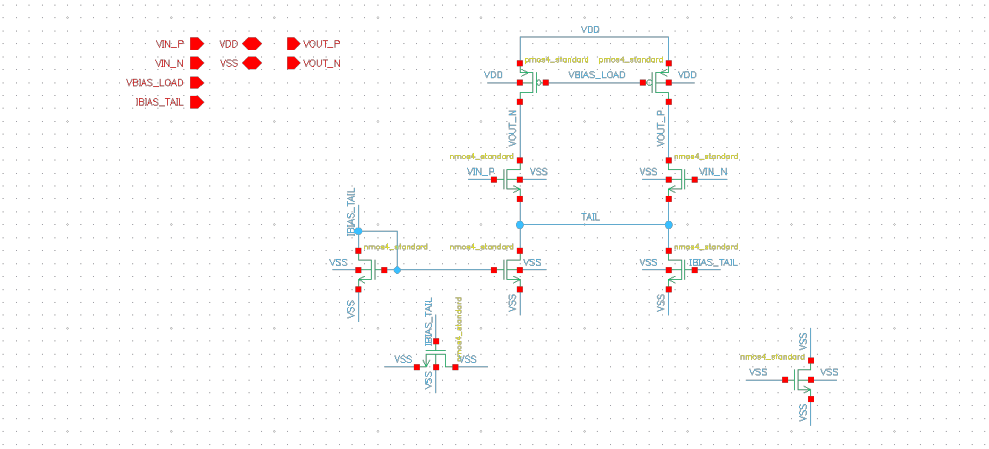


Figure 2.2: Differential amplifier schematic template example

This template holds only a human-readable description of the connections. The schematic will be copied over with actual values filled in by BAG afterward. One thing to note is the presence of seemingly useless transistors, like in the bottom right. These transistors are used by BAG to properly create dummy transistors in the layout, and anything else can be removed if not used in the schematic generator. Additionally, the transistor in the bottom left used for adding stabilization capacitance can also be removed if desired. Example layouts below will not have this transistor.

Using the schematic template, the user then decides how many rows of transistors will be in the layout, and assigns a number of transistors to each row. There are a number of helper functions to generate a data structure containing information about which row the transistor is in, the drain/source metal directions, number of fingers, etc. which is called the "initialization step". An example of how one might set up the rows and initialization for the amplifier in Figure 2.2 is shown in Listing 2.1 and Listing 2.2. Note that these code blocks are only portions of the full layout generator and there are, in general, a small number of extra lines required. This section highlights only the most important portions of a generator.

```
        # Rows are ordered from bottom to top
        # To use TrackManager, an ordered list of wiring types and their locations must be provided.
        # Define two lists, one for the nch rows and one for the pch rows
        # The lists are composed of dictionaries, one per row.
        # Each dictionary has two list entries (g and ds), which are ordered lists of what wire types will be present
        #  in the g and ds sections of that row. Ordering is from bottom to top of the design.
        wire_names = dict(
            nch=[
                # tail row
                dict(
                    g=['bias'],
                    ds=['bias']
                ),
                # Input row
                dict(
                    g=['sig'],
                    ds=['sig']
                )
            ],
            pch=[
                # top row
                dict(
                    ds=['sig'],
                    g=['bias']
                )
            ]
        )
        layout_helper.wire_names = wire_names
        # Set up the row information
        # Row information contains the row properties like width/number of fins, orientation, intent, etc.
        # Storing in a row dictionary/object allows for convenient fetching of data in later functions
        row_tail = layout_helper.initialize_rows(row_name='tail',
                                         orient='R0',
                                         nch_or_pch='nch',
                                         )
        row_input = layout_helper.initialize_rows(row_name='input',
                                          orient='R0',
                                          nch_or_pch='nch',
                                          )
        row_mirror = layout_helper.initialize_rows(row_name='top',
                                           orient='R0',
                                           nch_or_pch='pch',
                                           )

        # Define the order of the rows (bottom to top) for this analogBase cell
        layout_helper.global_rows = [row_tail, row_input, row_mirror]
```

Listing 2.1: Creating rows for transistors

```
        ##########################################################################################
        # 2:
        # Initialize the transistors in the design
        # Storing each transistor's information (name, location, row, size, etc) in a
        # dictionary object allows for convenient use later in the code, and also
        # greatly simplifies the schematic generation
        # The initialization sets the transistor's row, width, and source/drain net names
        # for proper dummy creation
        ##########################################################################################
        tail_l = layout_helper.initialize_tx(name='tail_l', row=row_tail,
                    fg_spec='bottom_tail',
                    deff_net='TAIL')
        tail_r = layout_helper.initialize_tx(name='tail_r', row=row_tail,
                    fg_spec='bottom_tail',
                    deff_net='TAIL')
        bias = layout_helper.initialize_tx(name='bias', row=row_tail,
                    fg_spec='bottom_bias',
                    deff_net='IBIAS_TAIL')
        in_left = layout_helper.initialize_tx(name='in_left', row=row_input,
                    fg_spec='bottom_in',
                    seff_net='TAIL', deff_net='VOUT_N')
        in_right = layout_helper.initialize_tx(name='in_right', row=row_input,
                    fg_spec='bottom_in',
                    seff_net='TAIL', deff_net='VOUT_P')
        top_left = layout_helper.initialize_tx(name='top_left', row=row_mirror,
                    fg_spec='top',
                    deff_net='VOUT_N')
        top_right = layout_helper.initialize_tx(name='top_right', row=row_mirror,
                    fg_spec='top',
                    deff_net='VOUT_P')
```

Listing 2.2: Assigning transistors to a row and initializing

```
        # Calculate positions of transistors
        # This uses helper functions to place each transistor within a stack/column of a
        # specified starting index and
        # width, and with a certain alignment (left, right, centered) within that column
        layout_helper.assign_tx_column(tx=bias, offset=col_mid, fg_col=fg_mid, align=0)
        layout_helper.assign_tx_column(tx=tail_l, offset=col_stack_left,
            fg_col=fg_stack, align=0)
        layout_helper.assign_tx_column(tx=in_left, offset=col_stack_left,
            fg_col=fg_stack, align=0)
        layout_helper.assign_tx_column(tx=top_left, offset=col_stack_left,
            fg_col=fg_stack, align=0)
        layout_helper.assign_tx_column(tx=tail_r, offset=col_stack_right,
            fg_col=fg_stack, align=0)
        layout_helper.assign_tx_column(tx=in_right, offset=col_stack_right,
            fg_col=fg_stack, align=0)
        layout_helper.assign_tx_column(tx=top_right, offset=col_stack_right,
            fg_col=fg_stack, align=0)
```

Listing 2.3: Assigning transistors to columns

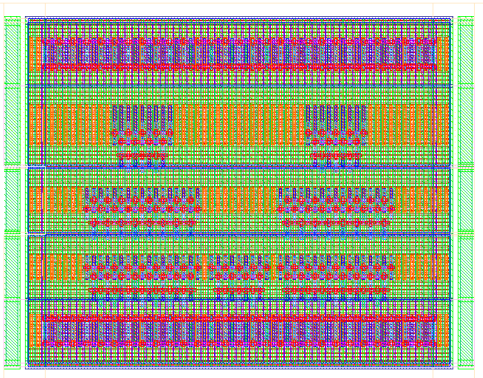
After initializing all transistors, the user then must specify their locations in the layout by creating fictitious columns that stacks of transistors will be placed in. Based on the number of fingers each transistor has, technology required spacing and any other spacing (to route dummies, etc.) the user can compute columns based on a number of fingers, and assign the transistors like in Listing 2.3.

The final transistor placement step is to set the drain and source orientations. A transistor's source can be routed up or down which affects where the gate placements are made, and the first diffusion region per transistor can be either source or drain to make alignment simpler. There is also a helper function to automatically compute based on number of fingers which region should be the source or drain based on another transistor the user wishes to align to. This is shown in Listing 2.4.

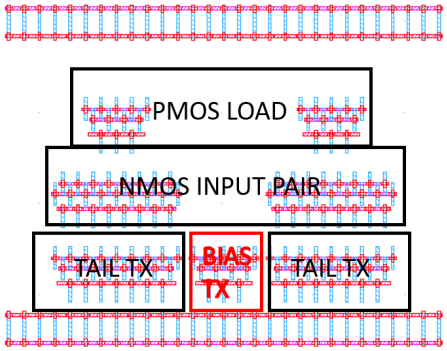
```
        ##########################################################################
        # 4:  Assign the transistor directions (s/d up vs down)
        #
        # Specify the directions that connections to the source and connections to the drain
        # will go (up vs down). Doing so will also determine how the gate is aligned
        # (ie will it be aligned to the source or drain)
        # See the bootcamp for more details
        # The helper functions used here help to abstract away whether the intended
        # source/drain diffusion region of a transistor occurs on the even or odd
        # columns of that device (BAG always considers the even columns of a
        # device to be the 's'). These helper functions allow a user to specify
        # whether the even columns should be the transistors effective source or
        # effective drain, so that the user does not need to worry about BAG's notation.
        ##########################################################################

        # Set tail bias tx  to have source on the leftmost diffusion (arbitrary)
        # and source going down
        layout_helper.set_tx_directions(tx=bias, seff='d', seff_dir=0)
        # Assign the input to be anti-aligned, so that the input source and tail
        # drain are vertically aligned
        layout_helper.set_tx_directions(tx=in_left, seff='s', seff_dir=0)
        layout_helper.set_tx_directions(tx=in_right, seff='s', seff_dir=0)

        layout_helper.assign_tx_matched_direction(target_tx=tail_l, source_tx=in_left,
            seff_dir=0, aligned=False)
        layout_helper.assign_tx_matched_direction(target_tx=tail_r, source_tx=in_right,
            seff_dir=0, aligned=False)

        layout_helper.assign_tx_matched_direction(target_tx=top_left,
            source_tx=in_left, seff_dir=2)
        layout_helper.assign_tx_matched_direction(target_tx=top_right,
            source_tx=in_right, seff_dir=2)
```

Listing 2.4: Determining transistor drain/source configurations

Finally, the difficult setup is complete, and the user can call the function `self.draw_base()` with the information about rows, transistors, etc. set up in the previous steps. BAG will then draw the all of the required polygons for the metal connections to the MOS devices and everything else. An example base layout with no wiring done of the schematic in Figure 2.2 is shown in Figure 2.3.



(a) Base layout of transistors with active regions, poly, etc.



(b) The same layout with only metal connection layers shown. Annotated with locations of transistors.

Figure 2.3: Base layout

```
        #Connect up bias gates + drain
        warr_bias_in = self.connect_to_tracks(
            [tail_l['g'], tail_r['g'], bias['d'], bias['g']],
            tid_tail_gate
        )
        #connect tail drains to input sources (tail node)
        warr_tail = self.connect_to_tracks(
            [tail_l['d'], tail_r['d'], in_right['s'], in_left['s']],
            tid_tail_ds
        )
```

Listing 2.5: Drawing wire connections.

After drawing the base layout, users can then specify how to connect wires and ports by specifying a metal layer, a wire width and a track. Tracks are an invisible grid that spans the design space used by BAG to place wires properly. Listing 2.5 shows an example command of how to connect elements. Note that this code makes no reference to anything specific nor does it "hardcode" any parameters. Everything is generic to the specified parameters. Figure 2.4 shows the connections made by BAG.

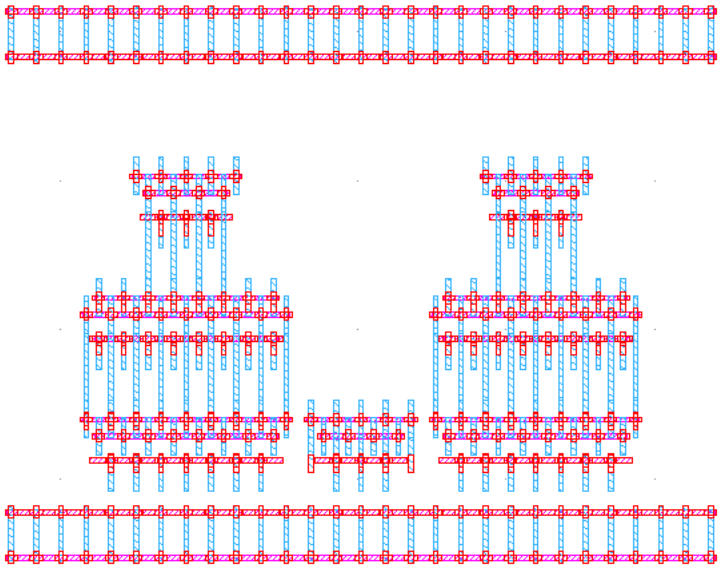


Figure 2.4: Transistor wiring

Lastly, the user finalizes connections and adds pins to wires. In the final step, BAG will draw dummy transistors and create straps for the power supplies, automatically routing the dummies' connections to the supply, like in Figure 2.5. With a finished layout generator, the user can now arbitrarily change their transistor specifications which will be reflected automatically in the wiring and sizing. Figure 2.6 shows the same generator with various width and number of finger choices. An important note is that the user can do all this

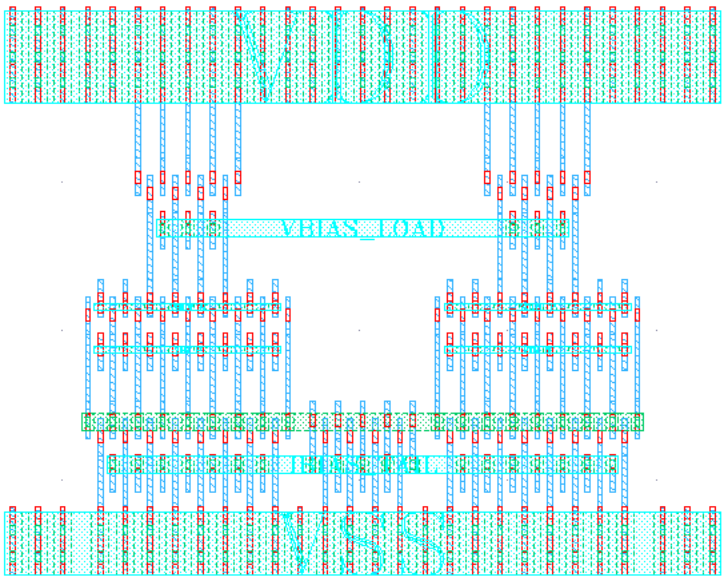
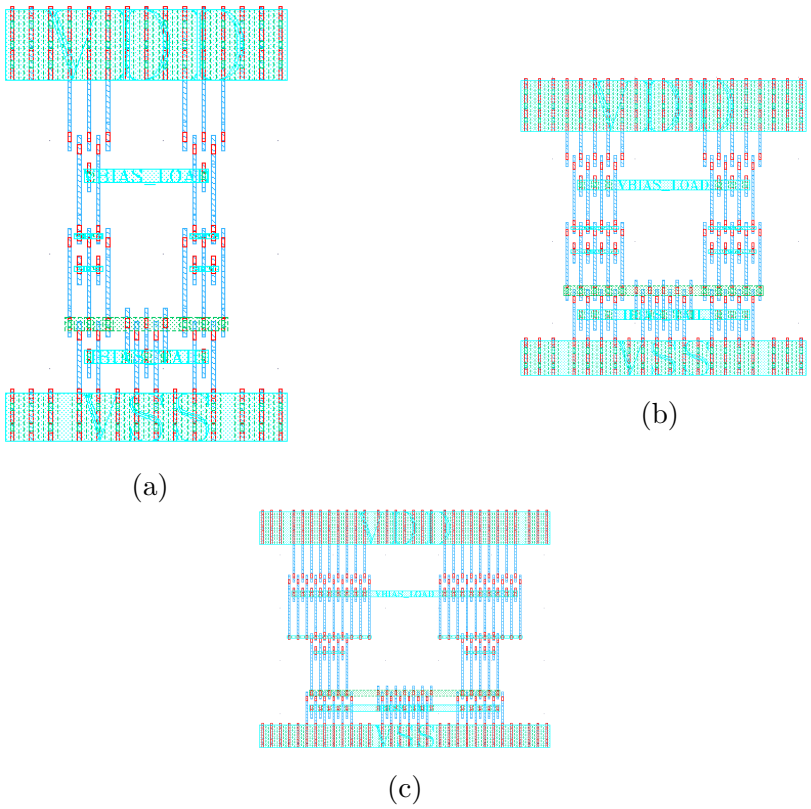


Figure 2.5: Finished layout (only metals shown)

without knowing any of the myriad of the layout design rules since BAG handles these complexities internally and abstracts them away from the user.



(a) (b) (c)

Figure 2.6: Differential amplifier layout with various widths and number of finger choices.

```
# Define the instance masters
  ...
        master_dtsa = self.new_template(params=dtsa_params,
            temp_cls=DoubleTailSenseAmplifier)
  ...
  ...
  ...
        x, y = layout_helper.get_on_track_offset(
            master_dtsa,
            x_offset=largest/2 + master_dtsa.bound_box.right_unit/2,
            y_offset=wiring_space + inst_combined.bound_box.top_unit,
            unit_mode=True
        )

        inst_dtsa = self.add_instance(
            master_dtsa, 'XDTSA',
            loc=(x, y),
            orient='MY',
            unit_mode=True
        )
```

Listing 2.6: Adding and placing templates to a layout

BAG also offers hierarchy in layout using `TemplateBase` [3]. When a library of smaller cells are created, the user can then "stamp" these cells into a larger unit and connect them together to form more complex systems. Listing 2.6 demonstrates the code to insert a double tail sense amplifier into a circuit. The user first creates a template, then computes a location where the circuit should be placed using helper functions to guarentee everything is aligned to the grid. Lastly, with the computed coordinates, the user can instantiate the circuit into the layout. An example of a circuit containing a differential transimpedance amplifier (TIA) and continuous-time linear equalizer (CTLE) is shown in Figure 2.7.

The benefit of codifying the layout procedure is that configuration can be automatically included. For example, should a designer want variable resistors in their circuit, they may opt for a resistor DAC. Resistor DACs are often an arbitrary number of arrayed unit resistors with digitally controlled switches. One method of implementing such a resistor DAC is a set of series resistors each with a parallel switch that can short out or connect any of the series resistors. Using `TemplateBase`, we can place any amount of template layouts which allows for a single generator with a large degree of freedom to create such a circuit. The user can choose how many bits, what type of switch to use (NMOS, PMOS, passgate), and even whether or not to include local inversion for the passgate. An example of multiple resistor DAC layouts of this style are shown in Figure 2.8 and a small portion of the configuration file in Listing 2.7.

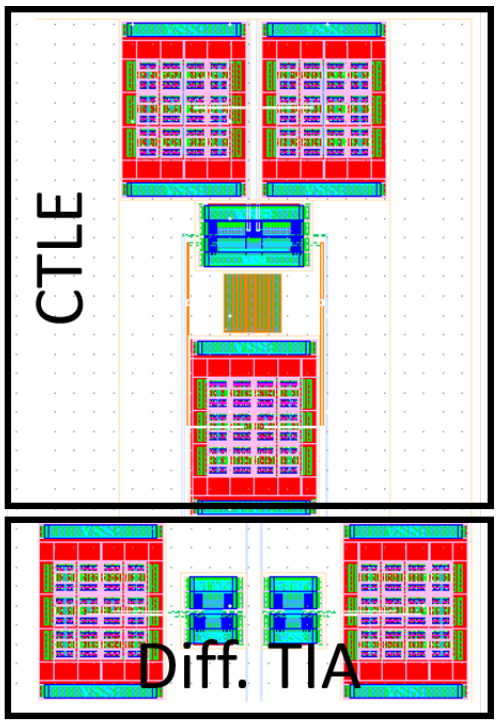
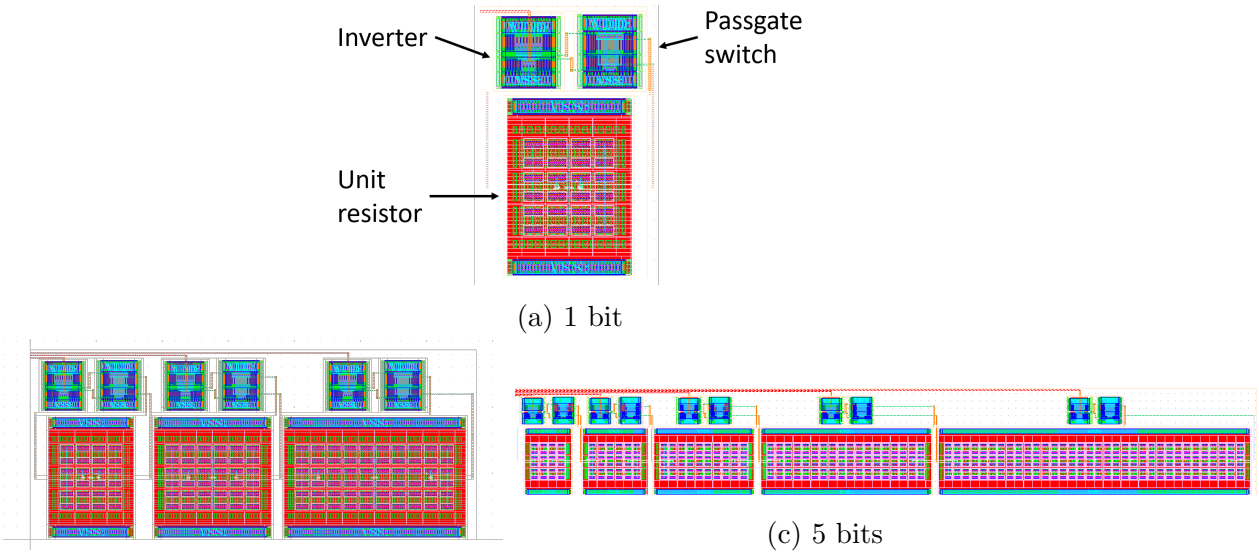


Figure 2.7: Differential TIA and CTLE

```
params:
  output_bits_dir: 'left' #where to drag the control bits to
  bits: 1
  switch_params:
    switch_type: 'transmission'
    include_inv: True
    switch_params:
      lch: !!float 14e-9
      guard_ring_nf: 0
      ptap_w: 6
      ntap_w: 6
      w_dict:  # Width of each row. Each row needs the width specified
        nmos:  6
        pmos:  12

      th_dict:  # Threshold information / thick ox / etc for each row
        nmos: 'standard'
        pmos: 'standard'

      seg_dict:  # Number of fingers of each transistor
        nmos: 8
        pmos: 16
...
...
...
res_params:
    show_pins: True
    l: 0.5e-6
    w: 1.0e-6
    sub_type: 'ptap'
    threshold: 'standard'
    nser: 1
    npar: 1
    ndum: 1
    port_layer: 5
...
...
...
```

Listing 2.7: Resistor DAC params sample



(a) 1 bit

(b) 3 bits

(c) 5 bits

Figure 2.8: Various resistor DAC generated layouts

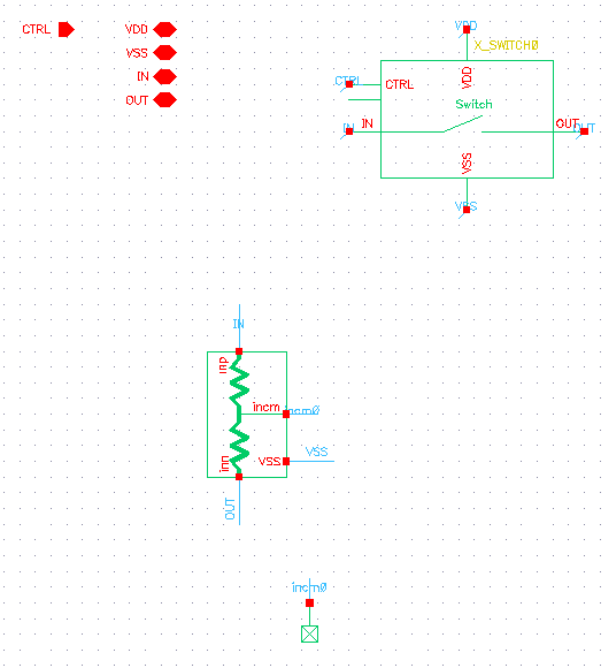
```
# Some pins are not needed depending on the type of switch used.
if switch_type == 'nmos':
        self.remove_pin('CTRL_B')
        self.remove_pin('VDD')
    else:
        if include_inv:
            self.remove_pin('CTRL_B')
        else:
            if switch_type == 'pmos':
                self.remove_pin('CTRL')
# When using a multi-bit resDAC, rename the input pin to have the required bitwidth.
    if bits > 1:
        if switch_type == 'nmos':
            self.rename_pin('CTRL', 'CTRL<' + str(bits-1) + ":0>")
        else:
            if include_inv or switch_type == 'transmission':
                self.rename_pin('CTRL', 'CTRL<' + str(bits - 1) + ":0>")
            if not include_inv:
                self.rename_pin('CTRL_B', 'CTRL_B<' + str(bits-1) + ":0>")
...
...
...
# Copy the unit cell many times with many names
self.array_instance('X_SWITCH0', switch_names, switch_port_names)
self.array_instance('X_RES0', res_names, res_port_names)
...
...
...
#Loop through all instances and use the design method to input the params from the spec file
     into the schematic
for ind, inst in enumerate(self.instances['X_RES0']):
  inst.design(**res_params_list[ind])
```

Listing 2.8: Resistor DAC schematic generator

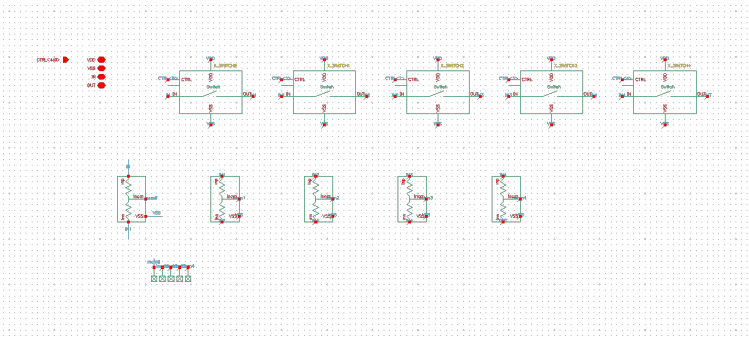
2.3 Schematic Generators

Schematic generators control the process of copying the template schematic mentioned previously and assigning values to the components based on the inputted spec file [3]. These generators have the capability of arraying and deleting instances, removing, adding or renaming pins and other simple operations.

Generally the user simply inputs commands to pass the component values into the design; but for circuits such as the resistor DAC in the previous section, the user can automatically array and resize components based on the layout like shown in Figure 2.9. Portions of the code that accomplishes this is shown in Listing 2.8.



(a) 1 bit



(b) 5 bits

Figure 2.9: Various resistor DAC schematics

2.4 Design Manager and Test Benches

As shown in Figure 2.1, design manager is a higher level entity to that of the layout and schematic generators [3]. Design managers contain within them multiple test bench managers that each have their own corresponding measurement manager. In the specifications file, the user decides which tests to run and design manager will generate an instance and pass it through the chosen simulations. This is the method used to test the circuits described in Chapter 4.

Tests are made by the user in a similar method to schematic generators. The user specifies a test bench setup with a generic DUT block and adds input sources and pins. Within a specifications file, the user describes how to wrap the inputs of the DUT to the generic block, as well as what outputs to save. These outputs are sent to the measurement manager where the user can, for example, compute 3dB frequencies, DC gain, etc. and save the results if desired. Multiple test benches can be run in one instance. A portion of an example parameters file for testbench management is shown in Listing 2.9.

```yaml
#Specifying how the ports of the device map to the generic DUT template
dut_wrappers:
  - name: 'ac_forward'
    lib: 'bag_testbenches_kh'
    cell: 'diff2SingleEnded_wrapper_ac_forward'
    params:
      dut_conns:
        IBIAS_TAIL: 'IBIAS_TAIL'
        VIN_N: 'VIN'
        VIN_P: 'VIP'
        VSS: 'VSS'
        VDD: 'VDD'
        VOUT_N: 'VON'
        VOUT_P: 'VOP'
#Choosing which measurement classes to run
measurements:
    #The - next to meas_type is not a mistake. It specifies that the entire contents of
    measurements is a list.
  - meas_type: 'CTLE_char'
    meas_package: 'verification_kh.CTLEMeasurementUnit'
    meas_class: 'CTLEMeasurementManager'
    out_fname: 'CTLE_char.yaml'
    sch_params: {}
    testbenches:
        ac_forward:
          tb_package: 'verification_kh.GenericACTB'
          tb_class: 'GenericACTB'
          tb_lib: 'bag_testbenches_ec'
          tb_cell: 'amp_tb_ac'
          wrapper_type: 'ac_forward'

    ...
    ...
    ...

    #Specify simulation variables
          fstart: !!float 1e3
          fstop: !!float 10e10
          fndec: 10
          sim_vars:
            ibias: !!float 900e-06
            vdd: !!float 0.8
            cload: !!float 20.0e-15
            vicm: !!float 0.6
    #Choose what outputs to save
          sim_outputs:
            outdiff: "VF(\"/vout\")"
            outcm: "VF(\"/vout_cm\")"
            indiff: "VF(\"/vin\")"
            ibias: "IDC(\"/VSUP/PLUS\")"
```

Listing 2.9: Test bench parameters

An example test bench manager, `generic_AC_TB`,[1] inputs an AC voltage or current and runs an AC simulation and noise simulation. The corresponding measurement manager sifts through the data and (regardless of the transfer function shape) computes the DC gain and overall bandwidth. The noise data is integrated and reported, as well as CMRR.

Since BAG is written in Python, users can easily extend or add features to BAG. An example is the extension of `DesignManager` to `SweepDesignManager`. This subclass inherits all the basic properties and functions in design manager, but allows the user to specify a set of variables in the parameter file to sweep. BAG will then automatically generate one instance per parameter value in the range, and simulate them all in parallel. This allows for the same type of sweeps one would do manually, but additionally includes parasitics and LVS.

[1] This test bench, and all others used in this report come courtesy of Kourosh H. of team Vlada.

Chapter 3

Example Generator Implementations

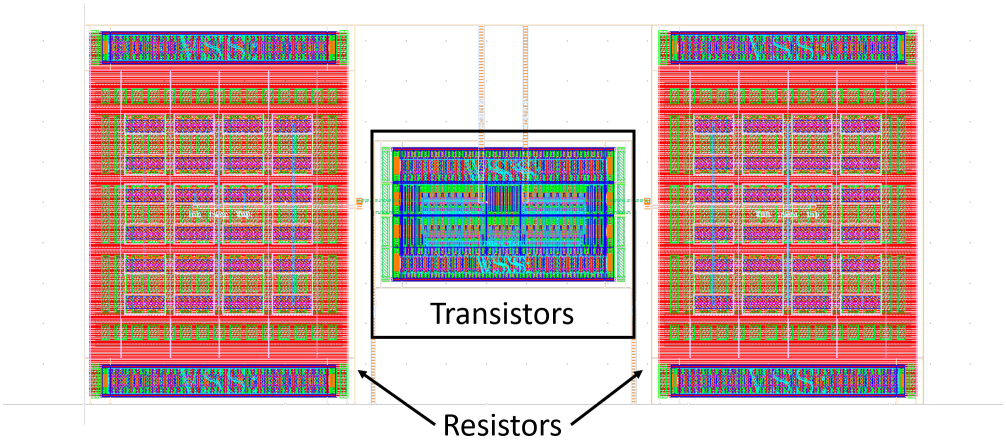
To illustrate a few possibilities of BAG generators, we show various circuits of increasing complexity.

3.1 Starting Small: Passive Loaded Differential Amplifier

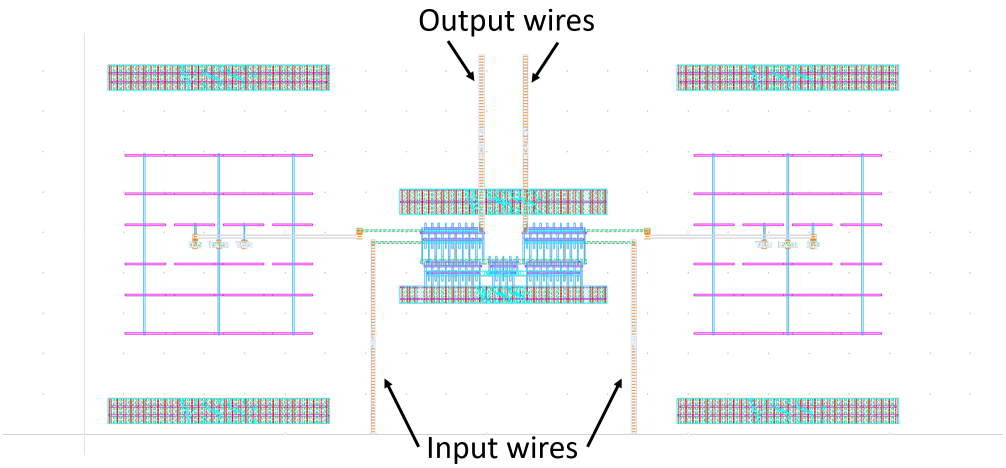
A very commonly used block in many ICs is the differential amplifier. In this case the load is passive. The topology is the same as that used in Chapter 4, and can be seen in Figure 4.7. An example generated layout instance is shown in Figure 3.1. All generators demonstrated have wire width parametrized so each type of wire (signal, bias, clock) can have a specific, customizable width through use of `TrackManager`. This class lets the user specify a width for each layer of interest and map them to a name. For example, if the user wants a specific width for all nets carrying a clock signal, then the user can specify the width all clock wires should be on each layer. Listing 3.1 shows a sample setup. When choosing a wire width to make a connection, the generator will select the width based on the passed specs, meaning wire widths can be parametrized as well. A section of code using this wire width is shown in 3.2.

This generator additionally allows the user to specify resistor unit cell sizes, number of parallel and series units and transistor dimensions (width, fingers) the same way as demonstrated in Chapter 2.

Another option for this generator is nearly identical, but with 3-bit resistor DACs instead of a single resistor as in Figure 3.2



(a) Full layout



(b) Only the metals

Figure 3.1: Differential amplifier layout

3.2 Getting Fancier: A Double Tail Sense Amplifier and Latch

Yet another crucial component in an analog receiver is the sampler. Before converting to purely digital processing, the received bits must be processed as either a 1 or a 0 through the use of a sampling circuit. While there are many ways to implement sampling, this report makes use of a double tail sense amp (DTSA) [4]. The schematic and operation are shown in Figures 4.7 and 4.8 respectively. An example layout is shown in Figure 3.3. The DTSA block exemplifies even more of BAGs capability to include customization. In addition to all previously mentioned parametrization (wire widths, transistor sizings, etc.) this block

```
tr_widths:
    # How wide to make the actual wires
    # format is {metal layer: track width, metal layer: width in tracks}
    bias: {4: 2, 5: 2}
    clk: {4: 1, 5: 1}
    sig: {4: 1, 5: 1}
  tr_spaces:
    # How wide to make the spaces between each wire of a certain type
    # same formatting. {metal layer: width in tracks}
    !!python/tuple ['bias', '']: {4: 1, 5: 1}
    !!python/tuple ['clk', '']: {4: 3, 5: 3}
    !!python/tuple ['clk', 'clk']: {4: 2, 5: 2}
    !!python/tuple ['clk', 'bias']: {4: 3, 5: 3}
    !!python/tuple ['sig', '']: {4: 2, 5: 2}
```

Listing 3.1: Track manager setup

```
# Create a track to put a wire on for connecting the resistor output
# terminal to the input terminal
        tid_extend_res_p_in_p = TrackID(
       #A way of specifying I want the next horizontal layer above my lowest horz. layer
         horz_conn_layer + 2,
         track_idx=self.grid.coord_to_nearest_track( #Helper function
             layer_id=horz_conn_layer + 2,
            #This represents the coordinate of the resistor port wire
             coord=(res_p_inp.get_bbox_array(self.grid).bottom_unit + \
                    diff_amp_vout_p.get_bbox_array(self.grid).top_unit)/2,
             unit_mode=True
         ),
        #This is where the parametrization comes in. By changing the width corresponding
        # to 'bias' in the spec file, the width will automatically change here.
         width=tr_manager.get_width(horz_conn_layer+2, 'bias')
    )
```

Listing 3.2: Track manager usage

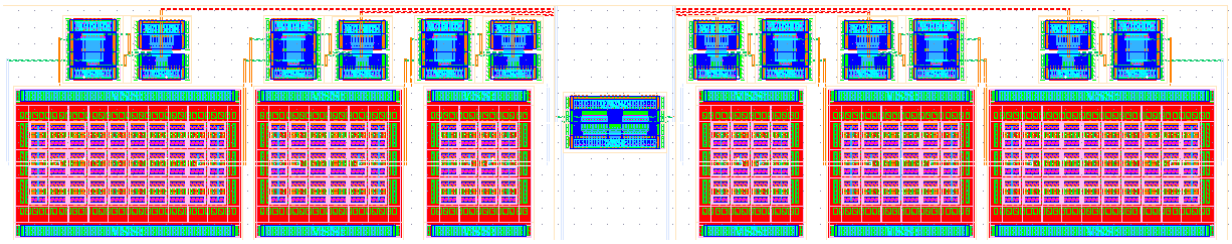
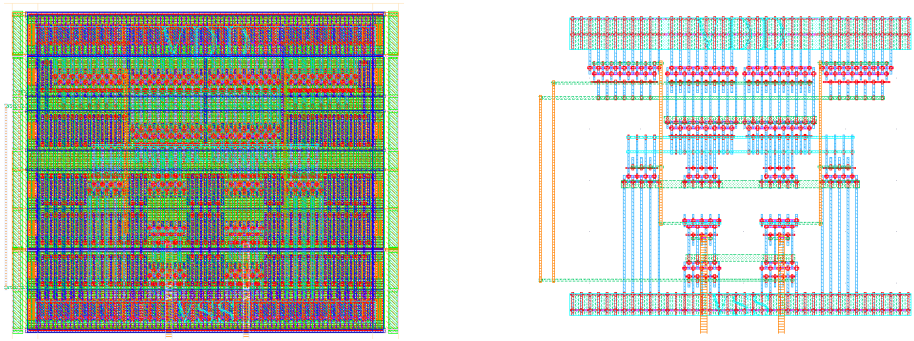


Figure 3.2: Passive diff-amp with resistor DAC

also includes an option to generate input pair offset correction circuits. These circuits are implemented as a current that subtracts from the input pair's current during the integration step of operation (discussed more in Chapter 4). The generator automatically accounts for how the setup changes when offset correction is included and automatically adds more pins/labels to the layout, as shown in Figure 3.4. Finally, the output of a comparator like this is only valid for a short time. We need a latch to store the value in between evaluation



(a) Full layout (b) Only the metals

Figure 3.3: DTSA layout

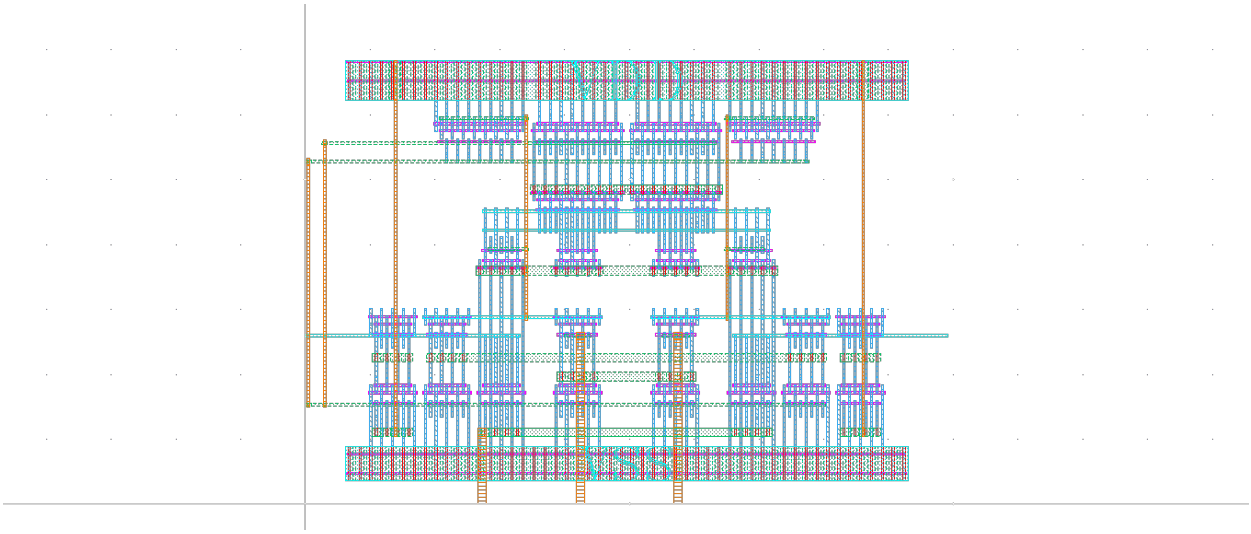


Figure 3.4: DTSA with offset correction

cycles. Thankfully, this is possible with BAG. Using TemplateBase we can attach a latch made previously to the output of the DTSA, like in Figure 3.5.

3.3 Endless Possibilities: An Entire Receiver Analog Front End Receiver or Two

Manually creating the layout for an entire analog front-end (AFE) of the reciever is a daunting process. As a final demonstration of how a user can start with small blocks and eventually create large, complex generators, two different front end architectures are shown. It is important to note that the size and complexity of these circuits are non-trivial, and would require significant manual work. These generators are capable of creating and extracting the

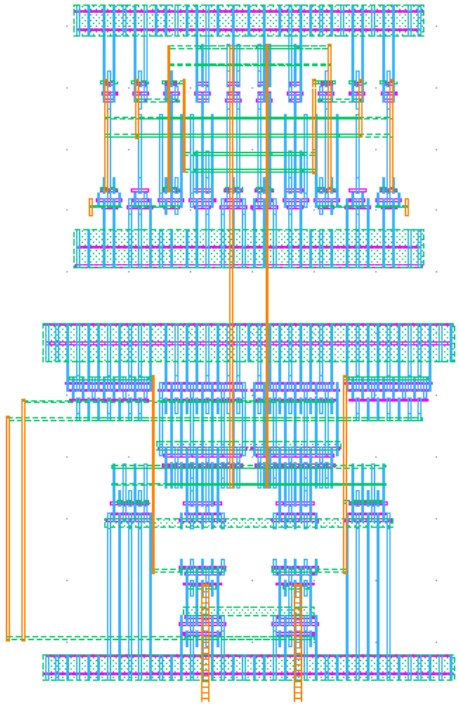


Figure 3.5: DTSA with latch

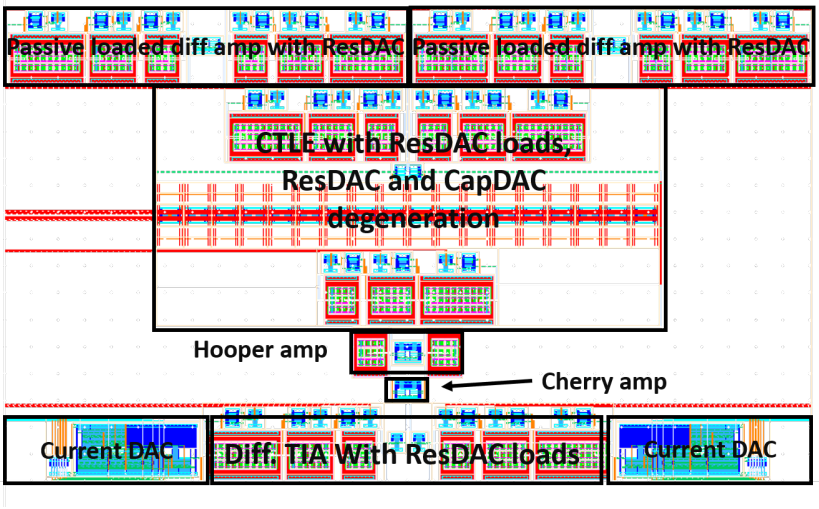
entire layout in seconds, which truly allows faster design iteration by removing the bottleneck almost entirely.

The first front end (Figure 3.6) is a chain of a TIA followed by a Cherry-Hooper amplifier stage, then a CTLE with resistor DACs and a capacitor DAC and two parallel preamplifiers also with resistor DAC loads. As will be discussed in Chapter 4, there are also current DACs to establish a common mode output in the first stage. The important thing to note about these generated layouts is how much control the user has over the dimensions of the blocks. Extremely wide or tall, or balanced layouts are possible. The aspect ratio is fully customizable, and can be changed at a whim with a simple rerun of the script.

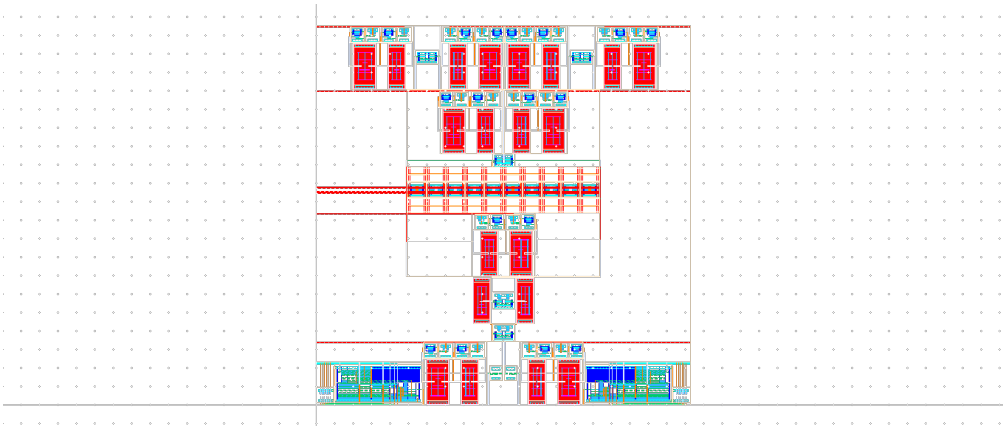
There are also versions that include DACs for every passive element as well as current DACs for every bias input. There are versions that remove the Cherry-Hooper stage and include a comparator. These various variants are also used for different simulation stages of design and verifiaction.

Another example of a front end is the one that will be used in Chapter 4. This AFE is a quad data rate AFE similar to the previous AFE and is comprised of the chain: TIA, CTLE, 2x parallel passive diff amps, 4x samplers. The architecture and operation is explained in Chapter 4. The layout is shown in Figure 3.7, and of course has the same degree of customizability alluded to in this entire chapter. The main point of demonstrating these layouts is that with a library of "leaf cells," composing large circuits is a very feasible task

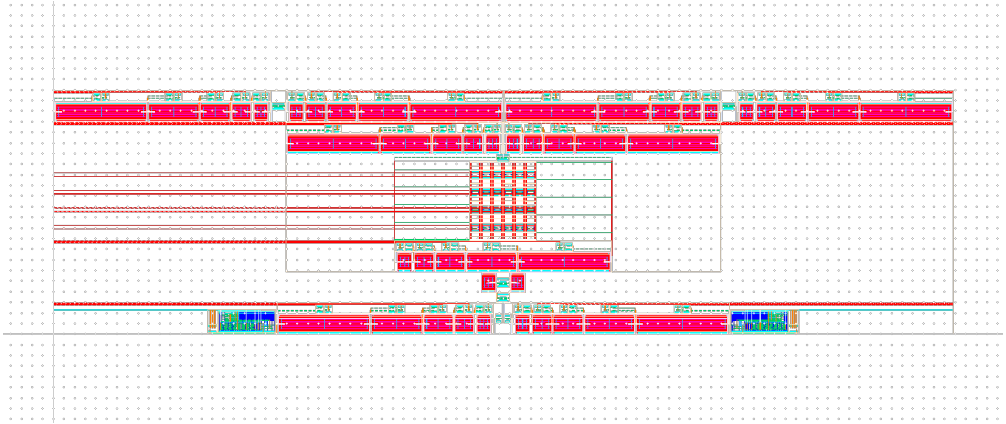
that would take an experienced user only one to two days to implement. With BAG, layout is no longer a painstaking process.



(a) AFE full layout, annotated with positions of blocks



(b) The same AFE with different parameters



(c) The same AFE with different parameters

Figure 3.6: Various generated AFE layouts

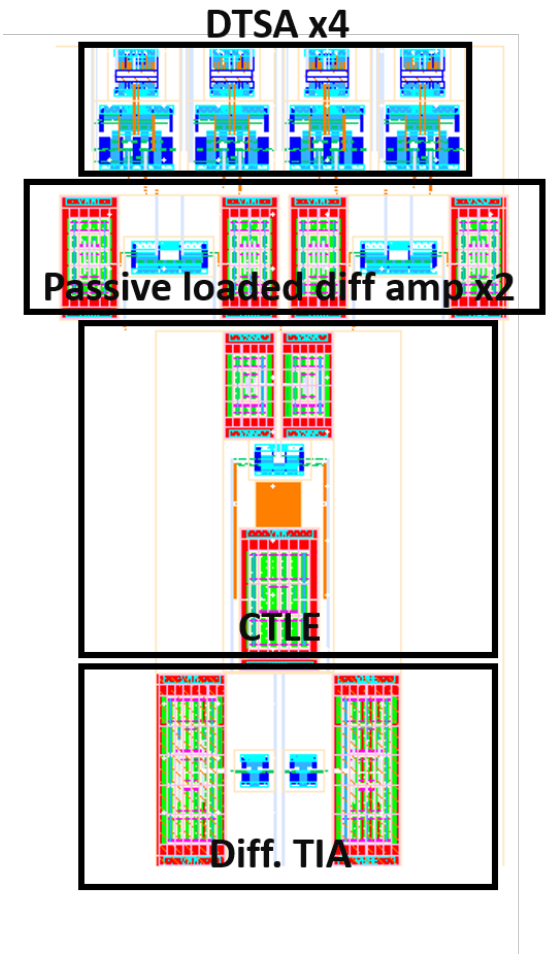


Figure 3.7: Second front end

Chapter 4

Design Problem: An Optical Receiver

Silicon photonics is a relatively new field, but the increasing possibility of incorporating photonics elements on chip with the electronics has led to optical circuits becoming a research hotspot.

To demonstrate the capabilities of BAG, an optical receiver design from concept to verification is shown.

4.1 Problem Statement and Architecture

Photonic communication systems transmit and receive data using light, meaning that the receiver element must be a photo diode. A reverse biased photo diode receiver element can be modeled as a current source with some capacitance in parallel. This is the "input", analogous to an antenna in typical RF receivers. For this design, the following arbitrary specs are required:

- 14nm FinFET technology
- A data rate of 25Gbps
- A photo diode capacitance of 50fF
- $40\mu A$ peak to peak photo diode current
- $V_{DD} = 0.8V$
- Must use BAG for generation and the majority of testing

There are no power constraints or architecture constraints with the condition that any chosen architecture will be implemented in BAG. We will assume the photonics are already implemented, and we will also not simulate for temperature, voltage or process variations. Each transistor will be of a unit size, and only the number of fingers will change. Noise in general should be low, but there is no strict value requirement. We will see in the "Future

Work" of Chapter 5 how this could be extended, and an example of how one could generalize the design procedure automatically.

Architecture Choice and Concerns

The architecture is based off the receiver in [10] and is shown below:

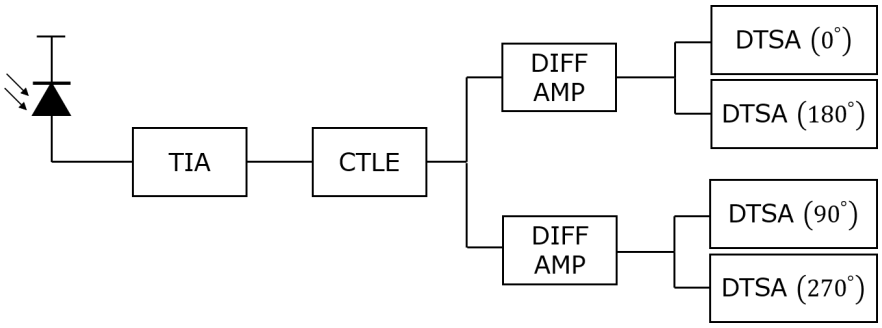


Figure 4.1: System architecture

This receiver is a quad data rate (QDR) receiver with each comparator operating in 90° phase offsets at a quarter of the clock rate to reduce the comparator constraints. Technically, only a single comparator clocked at 25GHz is necessary, however the time required for a comparator to decide between a 1 or 0 bit depends on the available voltage drive and hence impacts the receiver current sensitivity. To meet the target speed and sensitivity specifications, we use four comparators that operate only a quarter of the time to allow enough time for decision making and regeneration. This will be further discussed in following parts of this chapter.

Another design choice made is to drive the 0° and 180° offset comparators with the same preamp, and 90°and 270° together. This was done to reduce the effect of back-injection of the clock into the circuit elements before it.

The TIA is the main gain stage and is used to convert the incoming current waveform into a voltage. Since the TIA is the first block in the chain, the formula for cascaded noise figures

$$F_{total} = F_1 + \frac{F_2 - 1}{G_1} + \frac{F_3 - 1}{G_1 G_2} + ... \tag{4.1}$$

(where F_1 and G_i are the noise factor and gain of stage i respectively) tells us we want the TIA to be high gain and low noise in order to reduce the overall noise factor of the system.

The TIA is followed by a CTLE. A CTLE has a zero in its transfer function that can be placed at a specific frequency, which theoretically allows the designer to extend the bandwidth of previous stages as in Figure 4.2. One concern of the CTLE however, is that it

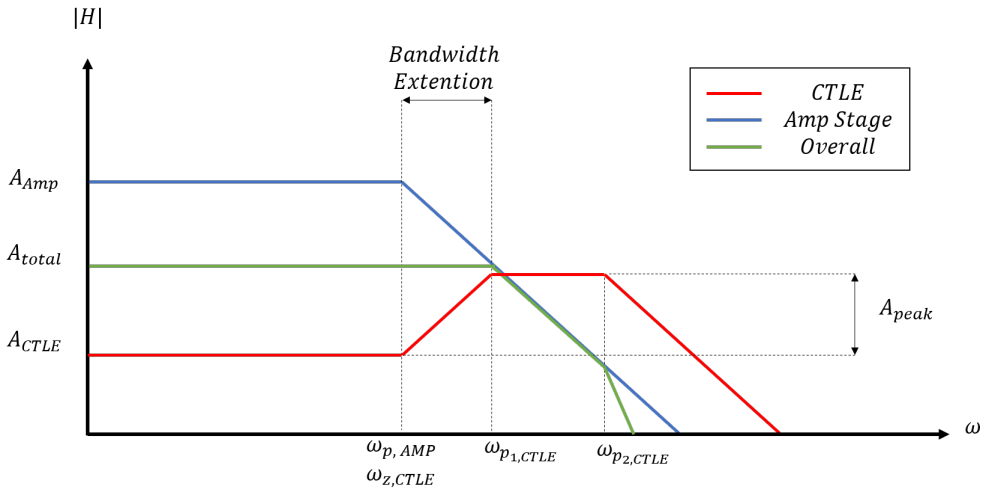


Figure 4.2: CTLE effect on a generic one-pole system

can only, in general, shape the energy of the frequency spectrum. This means that usually to increase the gain at high frequency, we must throw away DC gain, as shown below:

Since we are targeting 25Gbps, the empirical target bandwidth the front end receiver needs for a relatively optimal trade off in power and introduced ISI is $\approx 0.7\times$ the data rate, or roughly 17GHz [11]. The bare minimum would be roughly half, or 12.5GHz. We will target somewhere in between. A good rule of thumb for comparators is that they need milivolts of signal swing to consistently measure correctly. The gain bandwidth product of the TIA is unlikely to be large enough to get a $40\mu A$ signal to roughly $10mV$ across a frequency range of 0Hz to 17GHz and be low noise, so we increase the TIA gain and lower the bandwidth so that even with the CTLE DC gain reduction, we can still get decent gain in conjunction with the CTLE bandwidth extension.

The final stage before the comparators is a set of passively loaded differential amplifiers. Their purpose is threefold. The CTLE would have to simultaneously drive four comparators, which is a fairly large capacitive load. This limits the maximum achievable peaking gain by pushing in the second pole. Additionally, the comparators tend to inject their clock signal backwards which can impact the CTLE's operation periodically. The amplifiers serve as buffers which will isolate the kickback, and act as an intermediate step to reduce the amount of capacitance the CTLE has to drive. The DC gain is also expected to be too low due to the CTLE's reduction, so the amplifiers will provide a relatively small gain ($\approx 2\times$) to get as much swing as possible.

4.2 Transimpedance Amplifier

The TIA is implemented as a single-ended input with a dummy reference using two CMOS inverters with resistive feedback.

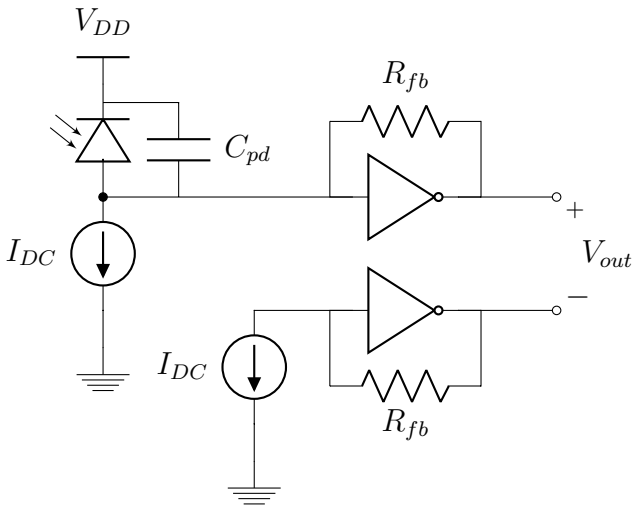


Figure 4.3: TIA Schematic

The purpose of implementing the TIA in such a fashion is partly due to the work in [10]. This project report began as a project to port the design in [10] into a more generic BAG style to demonstrate the process-portability of BAG. Additionally, [8] shows that this architecture can be used with a scheme that splits the photodiode differentially to take advantage of the dummy block. At DC, if $I_{DC} = 0$ then the input and output common mode voltages are equal. The purpose of the DC current source is to force the output common mode to be higher if desired. Since the output will sit about mid-rail, that may not be enough to bias the input of the CTLE, so we will need DC current through R_{fb} to force this difference.

If we assume the inverters are an amplifier with gain $-A_v$ then the input impedance can be found by using Miller's theorem.

$$Z_{in} = Z_{C_{pd}} || \frac{R_{fb}}{(1 + A_v)} \tag{4.2}$$

which simplifies to

$$\frac{\frac{R_{fb}}{1+A_v}}{1 + j\omega \frac{R_{fb}}{1+A_v} C_{pd}} \tag{4.3}$$

Thus the input pole should be roughly at

$$\omega_p = \frac{1 + A_v}{R_{fb} C_{pd}} \tag{4.4}$$

assuming that C_{pd} is the photodiode capacitance plus any TIA input and wiring capacitance lumped together. From the small signal model of one half of the circuit (shown in Figure 4.4, ignoring C_{pd}) we can derive the gain. KCL at the output node gives

$$\frac{v_{out}}{r_{on} || r_{op}} + (g_{mn} + g_{mp})(i_{in} R_{fb} + v_{out}) - i_{in} = 0 \tag{4.5}$$

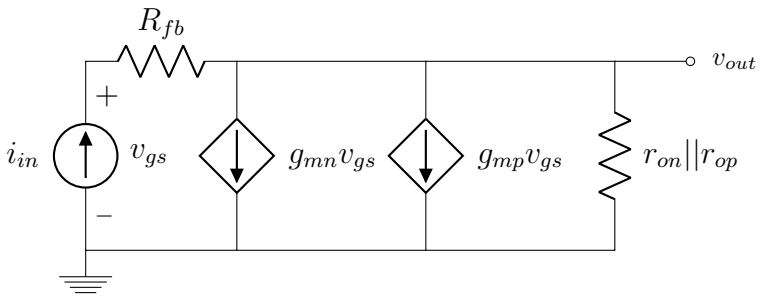


Figure 4.4: TIA low frequency small signal model

which simplifies to

$$\frac{v_{out}}{i_{in}} = \frac{R_{fb}(g_{mn} + g_{mp}) - 1}{-(g_{mn} + g_{mp}) - \frac{1}{r_{on}||r_{op}}} \tag{4.6}$$

If we assume R_{fb} and $r_{on}||r_{op}$ are both much larger than 1, then the transfer function simplifies to

$$|\frac{v_{out}}{i_{in}}| = R_{fb} \tag{4.7}$$

The gain is then approximately just R_{fb}. For a given choice of R_{fb}, we then use BAGs rapid iteration to sweep for transistor widths. As the size increases, device parasitics become dominant over the external capacitances, so there is an optimal size for bandwidth. Once the maximum bandwidth is found, this sets the device sizes. Since we know the CTLE can only get a couple of GHz of bandwidth extension, we can calculate and then sweep the TIA resistor to see what resistance gives a bandwidth of around 9GHz. If we assume the CTLE will cut the DC gain by roughly a third, and we can get about 1.5x amplification from the preamps, then the overall gain should still be high enough for the comparator.

To set the output common mode, we can assume the input will be around mid-rail, so the output can be approximated as follows:

$$\frac{V_o - \frac{V_{DD}}{2}}{R_{fb}} = I_{DC} \tag{4.8}$$

In order to bias the CTLE input, we want this to potentially be a little higher than midrail since the V_{GS} needs to be large enough to give headroom to the tail transistors. Plugging into equation 4.8 gives a starting point that can then be swept using BAG for better accuracy. Post-PEX hand-simulation is then used to fine tune the current the get the desired result. Note that by increasing the output common mode, the gain can reduce. This means that the resistance might need to be higher than anticipated. This is also determined by simulating BAG generated instances.

We were able to achieve a gain of 1500Ω with a bandwidth of 8.55GHz. In order to shift the common mode, I_{DC} was set to $55\mu A$.

4.3 Continuous Time Linear Equalizer

A CTLE is a simple amplifier, degenerated by a parallel resistor capacitor combination as shown below:

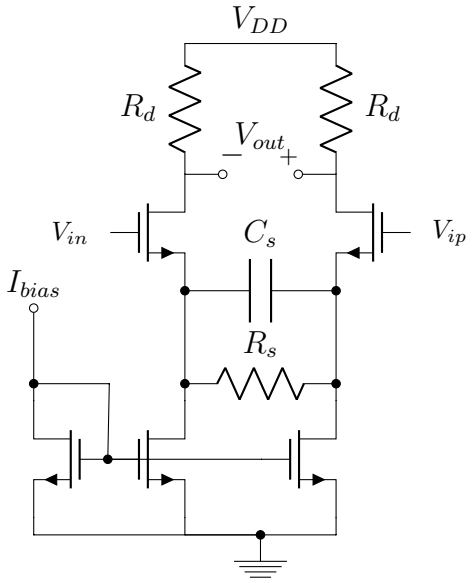


Figure 4.5: CTLE Schematic

Since we know the pole location of the TIA after simulation, we can design the CTLE to have its zero in close proximity. Firstly, we draw the approximate differential mode half circuit:

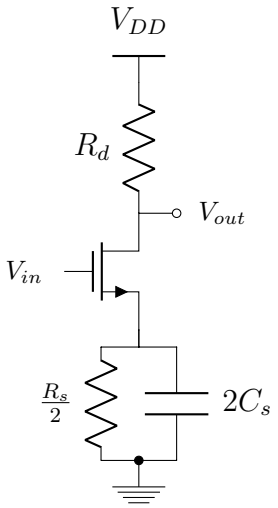


Figure 4.6: CTLE Half-Circuit

which by inspection, we know

$$G_m = \frac{g_m}{1 + g_m(\frac{R_s}{2}||\frac{Z_{Cs}}{2})} \tag{4.9}$$

$$G_m = \frac{2g_m(1 + j\omega R_s C_s)}{2 + g_m R_s + 2j\omega R_s C_s} \tag{4.10}$$

We can also determine the output impedance by inspection

$$R_o \approx R_d||\frac{Z_{C_l}}{2} = \frac{R_d}{1 + 2j\omega R_d C_l} \tag{4.11}$$

So the overall gain is then

$$G_m R_o = \frac{2g_m(1 + j\omega R_s C_s)R_d}{(2 + g_m R_s + 2j\omega R_s C_s)(1 + 2j\omega R_d C_l)} \tag{4.12}$$

which puts the zero at

$$\omega_z = \frac{1}{R_s C_s} \tag{4.13}$$

and the first pole at

$$\omega_{p1} = \frac{1 + gm\frac{R_s}{2}}{R_s C_s} \tag{4.14}$$

with the second pole at

$$\omega_{p2} = \frac{1}{2R_d C_l} \tag{4.15}$$

and lastly, the ideal peaking gain should be

$$A_{peak} = g_m R_d \tag{4.16}$$

From these equations, we can choose the values of the degeneration components and pullup resistors to place the zero and set the peak gain. We want the second pole to be far away at roughly 20GHz or more, which fixes R_d for a given C_l. If we want a peak gain of around 2, that then fixes the required g_m, which for an assumed V_{ov} of 200mV, fixes the bias current and transistor sizes. We will (quite conservatively) assume the input to each amplifier is 10fF, so the total load is 20fF. Through BAG, we can sweep component values near the desired points to get more accurate results.

Using the above steps, we were able to achieve a bandwidth extension of about 4GHz with a DC gain of roughly 0.6. When placed in succession to the TIA, the overall gain reduces to about 850 with a bandwidth extension to about 13.5GHz.

4.4 Preamplifiers

The preamplifiers are implemented as passively-loaded differential amplifiers as shown below. The main purpose is to serve as a buffer between the CTLE and the DTSAs. The DTSAs will provide a fairly substantial capacitive load to the CTLE as well as back-inject their clock, which is undesired. The preamplifiers ideally will have a gain of around 2, but we will target a gain greater than 1.

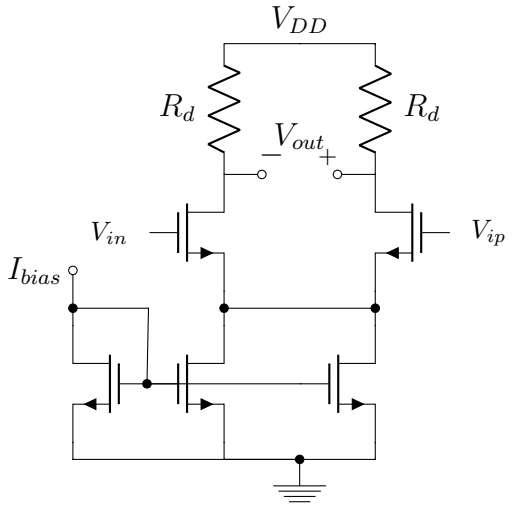


Figure 4.7: Preamp Schematic

Assuming the r_o of the transistors are fairly large, then the gain of this amplifier is known to be

$$A_v = g_m R_d \tag{4.17}$$

with the unity gain frequency at

$$\omega_u = \frac{g_m}{C_l} \tag{4.18}$$

Once we know the bandwidth of the CTLE, we can determine the required unity gain bandwidth for an overall gain of 2 at this frequency. Again assuming each DTSA provides 10fF of load, then we know C_l. This fixes the required g_m and therefore the device sizes and bias current.

Following these steps, the amplifiers were able to obtain a gain of 1.5 up to 13.5GHz. This sets the entire front end chain to have an overall bandwidth of 13.5GHz and a gain of 1200. Assuming a $40\mu A$ peak-to-peak input current swing, this should be plenty for the comparator.

4.5 Comparator

The comparator is implemented as a double tail sense amplifier.

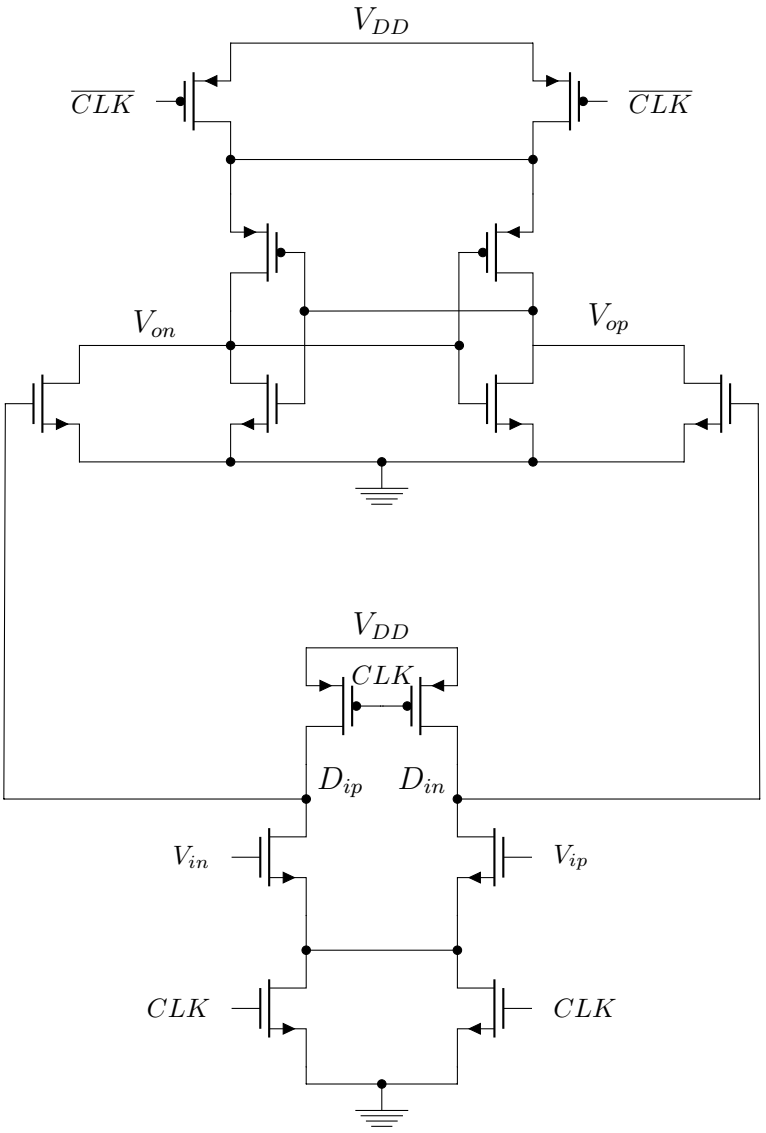


Figure 4.8: Double Tail Comparator Schematic

The DTSA consists of a clocked differential pair that feeds into cross-coupled inverters. Much like the classic StrongArm sense-amp [9], the DTSA first precharges the D_{in} and D_{ip} nodes up to V_{DD}. After CLK goes high, the input pairs activate and begin to drain charge from the parasitic capacitance at nodes D_{in} and D_{ip}. Depending on which input is larger, one side will discharge faster which will turn off one of the switches connected to those nodes faster than the other. This sets the value in the cross-coupled inverters which is then reinforced through positive feedback.

When CLK goes low, the circuit resets and precharges the D_{ip}/D_{in} nodes for the next cycle.

A typical cycle of the DTSA with a differential input of 10mV is plotted in Figure 4.9. The clock is generated by taking ideal clocks and passing them through a chain of inverters to generate both the real CLK and $\overline{CLK}$ signals.

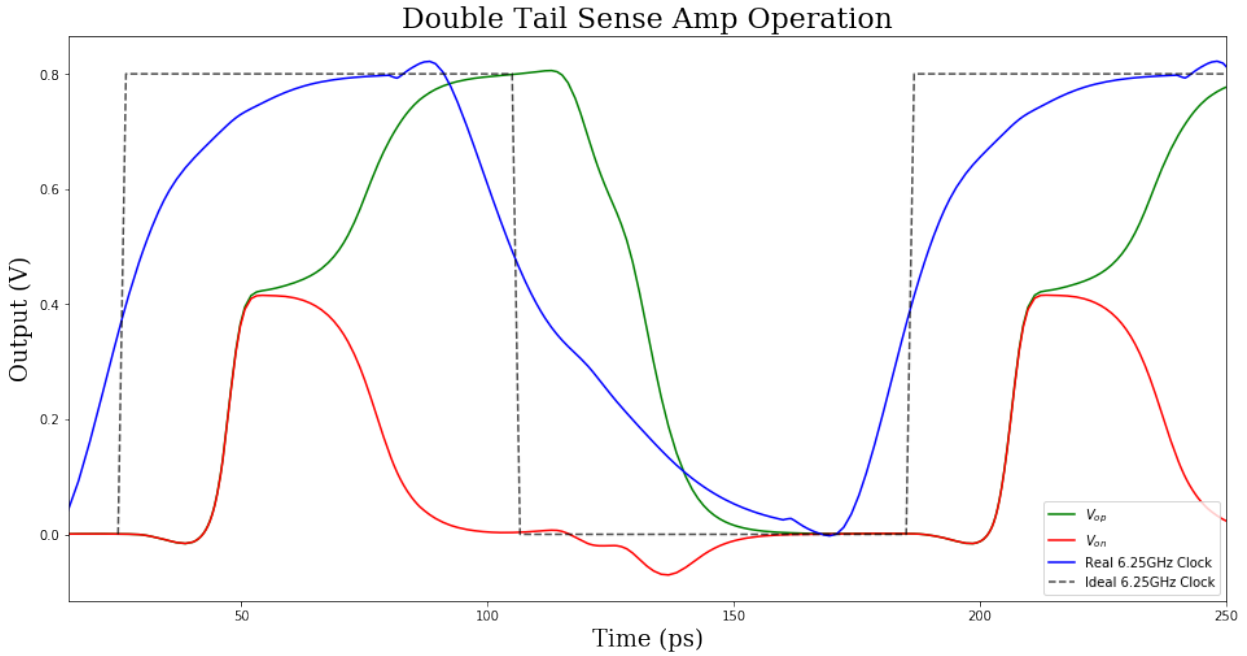


Figure 4.9: DTSA Typical Operation

The comparator usually is the first thing one would design, as its limitations generally set the required gain and noise specs for the front end. In this project, the comparator was designed last, since there is no power constraint. The rate at which voltage is reduced from D_{ip}/D_{in} is based on the current through the input pairs. By increasing the size of the input pairs, we can create a voltage difference much more quickly (assuming there is no process offset in the input pairs) which allows even small inputs to be sensed before the cycle ends. Thus, as long as our input is on the order of milivolts, we should be able to sense this.

As a test, we determine what the approximate noise tolerance of the DTSA is. We input a small differential signal (about 1mV) and run a transient simulation with noise over many

cycles. The fraction of incorrect evaluations to the total allows us to approximate σ_n, or the standard deviation of the noise tolerance for the DTSA. The results of this simulation are shown in Figure 4.10. With a differential input of 1mV, only one in 100 inputs failed.

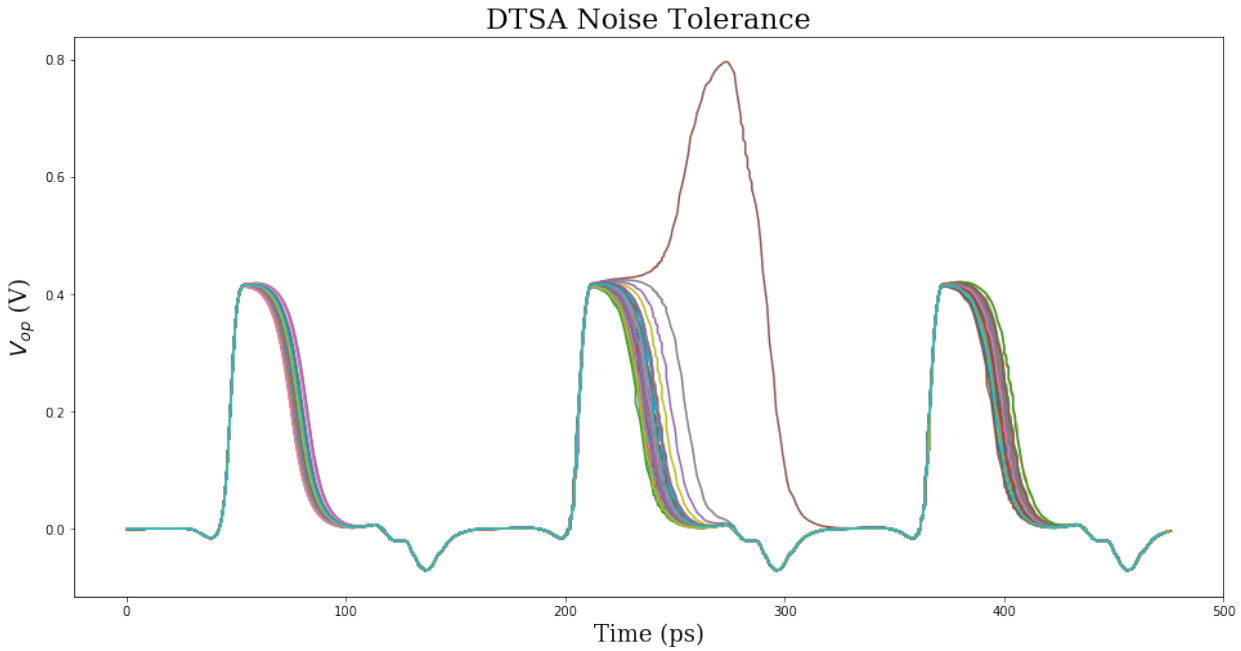


Figure 4.10: DTSA Noise

This implies that ±0.5mV input is approximately $3\sigma_n$. Although this includes both static overdrive and noise contribution, a previously run automated overdrive test determined that the static overdrive needed is much smaller than this input. This means that we can assume the majority of the input is the effect of noise. Thus, we should budget about $2 \times 9\sigma_n$, or ±3mV of the eye opening to noise tolerance for the DTSA for a BER of 10^{-12}. This, of course, ignores process variation. Process variation introduces an offset that affects the current in each branch. From a Monte Carlo simulation, one can find the average offset which can then additionally be added to the eye opening budget. There are also topologies that can correct for offset, which would not completely eliminate the issue, but would certainly reduce it.

4.6 Design Verification

There were four simulations done to verify the behavior of the entire receiver. Firstly, a simple AC simulation of the front-end chain up to the samplers in order to see if the circuit provides enough gain. This is shown in Figure 4.11

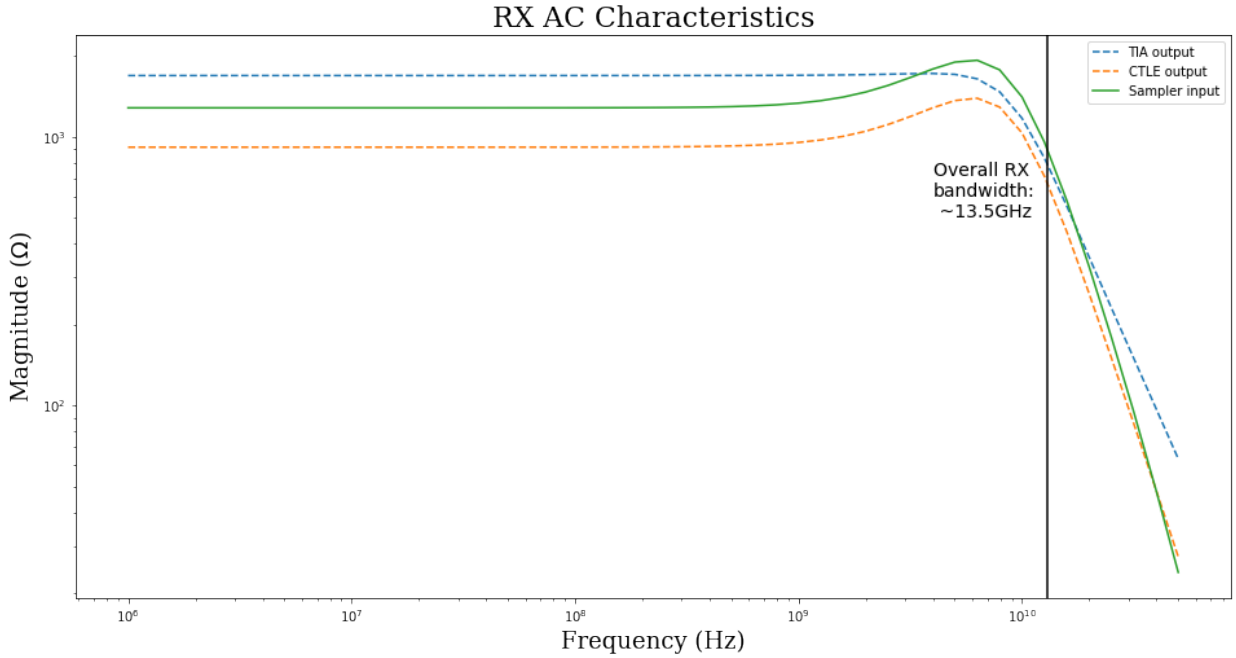


Figure 4.11: AFE AC Performance

To determine how much noise the amplifier chain introduces, an AC noise simulation was also run. The output power spectral density is plotted in Figure 4.12.

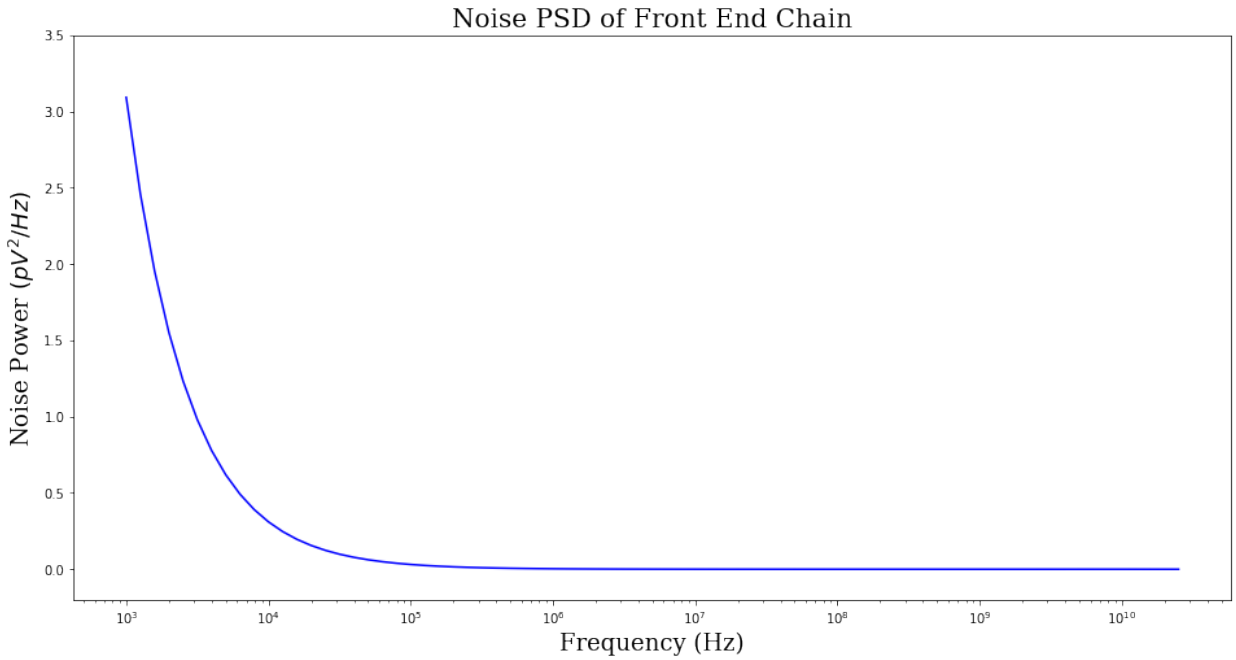


Figure 4.12: Front End Noise PSD

Using Python to integrate this function up to 25GHz gives an rms noise of $4.6\mu V^2$, so the mean expected noise is then 2mV.

From the previous sections, we know that the minimum input is approximately 1mV with a noise tolerance of 3mV for a BER of 10^{-12}. For an input of $40\mu A$ peak-to-peak, we achieve a gain of 1200 which puts the input amplitude at roughly 50mV peak-to-peak. This should be plenty. Unfortunately, the desired bandwidth was not met, although the bandwidth does surpass the theoretical minimum. This will mostly affect ISI performance, which will be seen in the next simulation.

To determine over many cycles how the AFE performs, we measure the eye diagram at the sampler inputs. This is shown in Figure 4.13. As can be seen, there is a fairly large eye

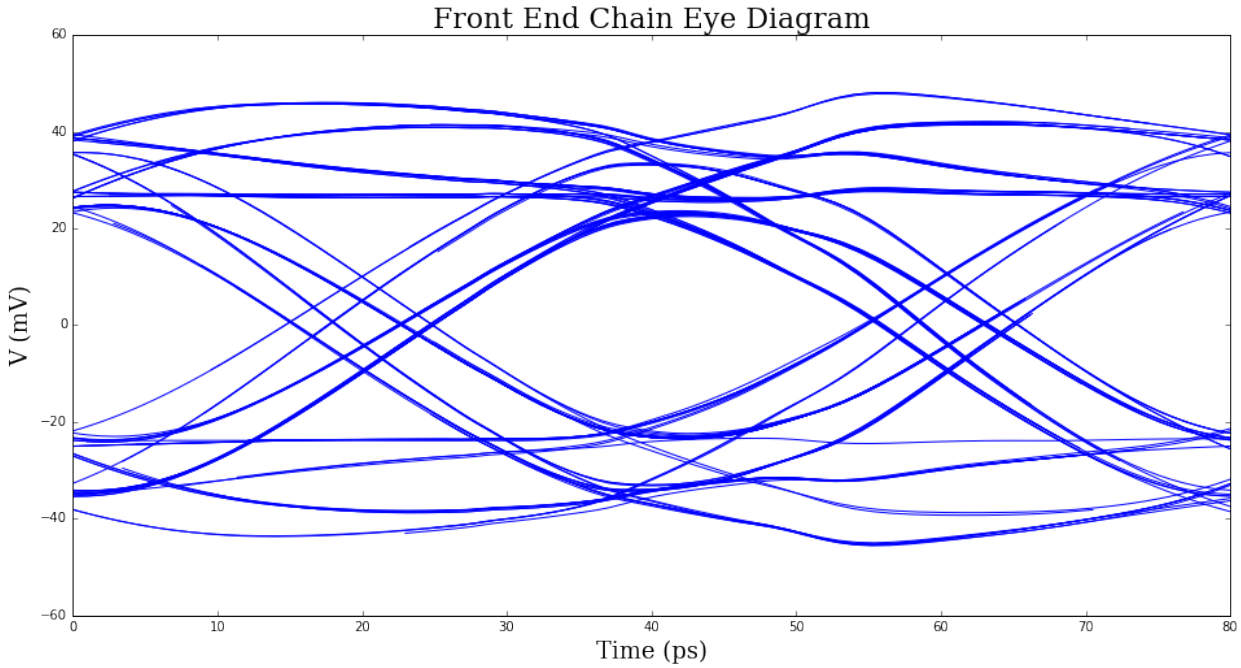


Figure 4.13: Sampler Input Eye Diagram

opening which surpasses the minimum required swing for proper evaluation.

Lastly, a PRBS32 pattern is fed into the receiver, and the sampler outputs are monitored in a transient simulation. Note that this architecture would not operate sufficiently due to the fact that the waveforms need to be adjusted in time to center the eye opening with the clock edges. This adjustment was done manually for this test. Also, since each sampler is only sampling once every 4 bits, the outputs are sliced and manually stitched together. A portion of the test input and output are shown in Figure 4.14. The patterns are clearly identical with only a time offset through the front end chain.

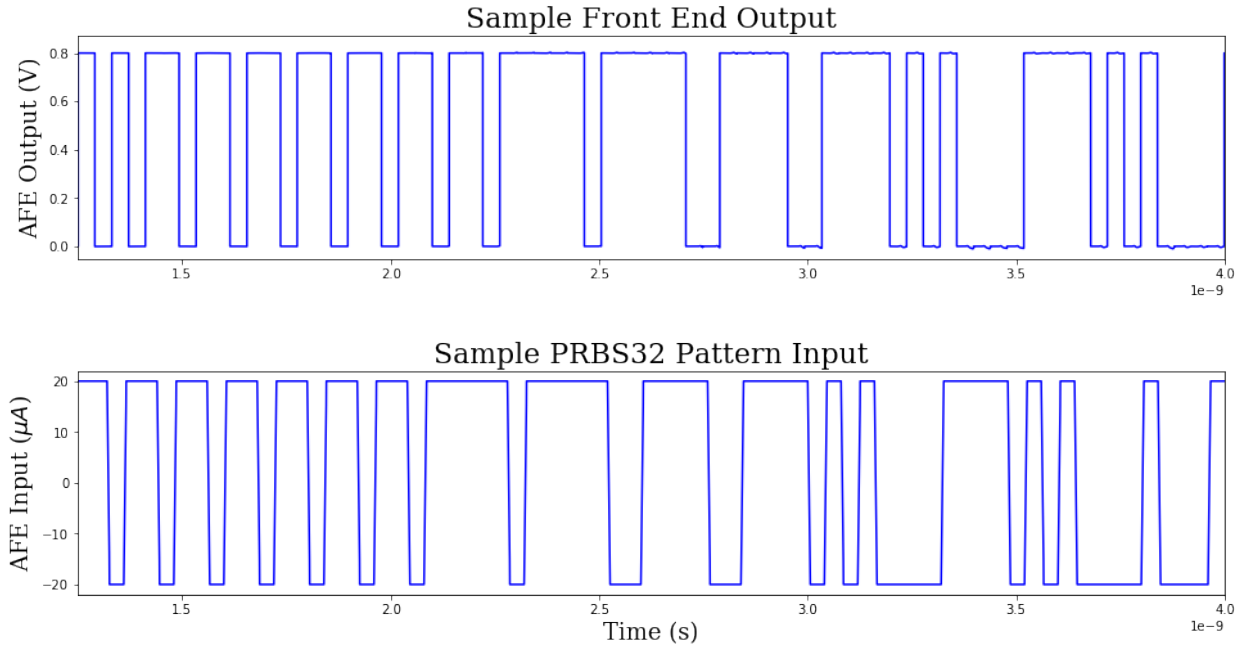


Figure 4.14: PRBS Pattern Response

As a final summary of the design, Table 4.1 shows the tabulated performance of the design. Measurements for energy and power are for the entire circuit, including biasing, clock buffers and the front-end up to the samplers. The measures were collected by simulating the transient current waveform and using Python to find the time average. The time average multiplied with VDD gives the average power consumption, which is then multiplied by the bits/time of the PRBS32 pattern input to get energy/bit.

Spec	Goal	Achieved
Area	-	$3106\mu m^2$
Energy/bit	-	0.3pJ/Bit
Peak Power	-	10.8mW
Avg. Power	-	7.52mW
Eye Opening	± 5mV	± 20mV
Bandwidth	17.5GHz	13.5GHz
Gain	1000Ω	1200Ω

Table 4.1: Performance Summary

Chapter 5

Conclusions, Future Work and Other Possibilities

And here we are at the finale. For those who made it this far, congrats! To those who read only the abstract before this, welcome. I hope it's interesting!

5.1 How BAG Saved the Day

A good question to ask is how BAG improved the process, if at all? The classic strategy would be to design with only some regard to layout parasitics, and simply hope the effect is not too great. Should the effects prove to be too substantial, modify the design and iterate until a solution is found. Does BAG fix this problem? The generators had to be written first which is clearly a non-trivial task. What time is even saved? What is the benefit over the old way?

Writing a generator is indeed a non-trivial task, however it only needs to be done once. A well-written generator will take any inputs (within reason) and generate a verified circuit with layout in seconds. An experienced BAG user can plausibly create a complicated generator in one to two days, which essentially permanently removes layout from the design iteration. Rather than spend two days on layout each time a change is made, a large time investment is made upfront for massive time savings in the future. BAG truly allows a designer to close the analog loop faster by removing the bottleneck from layout. If a user was provided a library of generators, the time savings are even greater. With a set of verified generators, the guesswork is completely taken out of the design process. Previously, parasitics either had to be estimated or ignored, and then accounted for afterwards. Now, parasitics can be directly included since the layout is essentially "free."

With BAG, the author was able to design, generate and (partially) test a fully LVS/PEX verified receiver front end working part-time in roughly two weeks. With an architecture chosen, only the sizing remained. Since layout generation was not an issue, seeing the effects of parasitics simplified the design process since they could be included automatically. Using

`SweepDesignManager` mentioned in Chapter 3, it becomes very easy to fine-tune parameters through sweeping automatically, since BAG will run PEX/LVS and simulate many designs in parallel. This way, the trade offs between component value choices are very clear and quick to see. Furthermore, no one had to painstakingly draw the layout by hand, massively improving life quality.

5.2 Future Work

Unfortunately, this design example is not finished. The design is only tested in the typical case at room temperature. A "tapeout ready" design would additionally require tests across temperature, and at process corners with power supply variation, etc. Furthermore, while layout effects are included, component offset is also substantial (such as in the sampler input pair) but is ignored. To include component offset, the topology of the sampler would certainly have to change, but this would require a potential redesign. Package parasitics are also ignored. As this was purely an example to demonstrate what is possible and how BAG can shorten the design process, many of these were ignored, but should be tested.

One massive reason to use BAG is the process portability. Theoretically, all generators presented should work in any process, but there will always be edge cases. These generators were only tested mainly in a 14nm process, and partially in a 45SOI process. Using design manager, hundreds of instances of various parameter choices were tested, but there are likely still issues that can arise with the right parameters. The generators can always be improved, and would be an excellent place to focus more work.

Lastly, a demonstration of how characterization can also be simplified would be a major focus. Testing is often the slowest part of a design, as the designer needs to set up and run numerous different tests and verify that the testing procedure is correct. Some test benches were mentioned in Chapter 3, but only some tests are implemented. For this project, all comparator testing was done by hand. A script that takes a design and runs it through many tests, post processes the data and returns a human-readable set of specs and plots for comparators would be highly desirable.

5.3 Related Work and Other Possibilities

Unsurprisingly, a platform like BAG can do even more that described in this report. One example is to fully remove the designer from the equation with a design script. The basis of these are discussed in [3]. Once the generators and test benches are made, the user can code the math and design procedure into a script that automatically computes device parameters and can automatically iterate and change values based on results. Within team Vlada at UC Berkeley, there is a substantial effort towards this methodology. At the time of writing, a script to design a front end very similar to that presented in Chapter 4 is being worked on and tested.

As is to be expected, BAG's rapid iteration opens the door for machine learning. BagNet in [6] demonstrates that BAG generators can be used as a tool to solve a constrained optimization problem with evolutionary algorithms coupled to deep-neural network discriminators using some of the generators demonstrated in this very work. BagNet shows the feasibility of designing complex analog/mixed signal circuits with sample-efficient, unsupervised learning by taking advantage of BAG's rapid iteration abilities. The author also compares the performance of BagNet solutions to a design script written by an expert designer.

Bibliography

[1] Phillip Allen. "THE PRACTICE OF ANALOG IC DESIGN". en. In: (2004), p. 23.

[2] *Amdahl's Law*. URL: https://home.wlu.edu/~whaleyt/classes/parallel/topics/amdahl.html (visited on 04/09/2019).

[3] Eric Chang et al. "BAG2: A process-portable framework for generator-based AMS circuit design". en. In: *2018 IEEE Custom Integrated Circuits Conference (CICC)*. San Diego, CA: IEEE, Apr. 2018, pp. 1–8. ISBN: 978-1-5386-2483-8. DOI: 10.1109/CICC.2018.8357061. URL: https://ieeexplore.ieee.org/document/8357061/ (visited on 04/03/2019).

[4] P. Chiu, B. Zimmer, and B. Nikolić. "A double-tail sense amplifier for low-voltage SRAM in 28nm technology". In: *2016 IEEE Asian Solid-State Circuits Conference (A-SSCC)*. Nov. 2016, pp. 181–184. DOI: 10.1109/ASSCC.2016.7844165.

[5] Scott Elder. *The REAL Cost for a Custom IC*. URL: https://www.planetanalog.com/author.asp?section_id=526&doc_id=559840 (visited on 04/09/2019).

[6] Koroush Hakhamaneshi et al. "Late Breaking Results: Analog Circuit Generator based on Deep Neural Network enhanced Combinatorial Optimization". In: *Design Automation Conference* (). (Accepted for publication in DAC 2019, Las Vegas, NV, June 2-6, p. 2.

[7] Nuno Lourenço, Ricardo Martins, and Nuno Horta. "Layout-Aware Sizing of Analog ICs using Floorplan & Routing Estimates for Parasitic Extraction". en. In: *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2015*. Grenoble, France: IEEE Conference Publications, 2015, pp. 1156–1161. ISBN: 978-3-9815370-4-8. DOI: 10.7873/DATE.2015.0411. URL: http://ieeexplore.ieee.org/xpl/articleDetails.jsp?arnumber=7092562 (visited on 04/09/2019).

[8] Nandish Mehta et al. "A 12Gb/s, 8.6uApp input sensitivity, monolithic-integrated fully differential optical receiver in CMOS 45nm SOI process". en. In: *ESSCIRC Conference 2016: 42nd European Solid-State Circuits Conference*. Lausanne, Switzerland: IEEE, Sept. 2016, pp. 491–494. ISBN: 978-1-5090-2972-3. DOI: 10.1109/ESSCIRC.2016.7598348. URL: http://ieeexplore.ieee.org/document/7598348/ (visited on 04/10/2019).

[9] Behzad Razavi. "The StrongARM Latch [A Circuit for All Seasons]". en. In: *IEEE Solid-State Circuits Magazine* 7.2 (2015), pp. 12–17. ISSN: 1943-0582. DOI: 10.1109/MSSC.2015.2418155. URL: http://ieeexplore.ieee.org/document/7130773/ (visited on 04/10/2019).

[10] Krishna Settaluri. "Photonic Links – From Theory to Automated Design". PhD thesis. EECS Department, University of California, Berkeley, Apr. 2019. URL: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2019/EECS-2019-8.html.

[11] Krishna T. Settaluri et al. "First Principles Optimization of Opto-Electronic Communication Links". en. In: *IEEE Transactions on Circuits and Systems I: Regular Papers* 64.5 (May 2017), pp. 1270–1283. ISSN: 1549-8328, 1558-0806. DOI: 10.1109/TCSI.2016.2633942. URL: http://ieeexplore.ieee.org/document/7807207/ (visited on 04/09/2019).