Internship Period:15/05/2023-14/07/2023

Internship Report on

Automation of Analog IC design

Department of Electrical Engineering

Supervisor: Dr. Abhishek Kumar

INDIAN INSTITUTE OF TECHNOLOGY HYDERABAD

भारतीय प्रौद्योगिकी संस्थान हैदराबाद

Indian Institute of Technology Hyderabad

Shailendra Mishra

INDIAN INSTITUTE OF TECHNOLOGY PATNA

Table of Contents:

Acknowledgment:

First and foremost, I would like to express my heartfelt gratitude to Indian Institute of Technology Hyderabad for providing me with such an invaluable internship experience through Summer Undergraduate Research Exposure 2023 (SURE-2023).

I am immensely grateful to Dr. Abhishek Kumar, my internship supervisor, for his valuable guidance, support, and mentorship throughout this journey. His expertise, feedback, and willingness to share knowledge have been instrumental in building up my understanding of Analog IC design and its shortcomings.

His invaluable guidance will serve as a guiding light throughout my future career, illuminating my path and encouraging me to take up challenges for the future benefit of society.

-Shailendra Mishra

Abstract:

We are aware of the fact that generating an Analog Layout requires much manual intervention and consumes a lot of time when compared to a Digital Layout. The main reason turns out to be the inability to design abstract blocks in the case of an Analog system when compared to the digital system. These abstract blocks are the ones that facilitate the process of automation. But that does not mean automating the analog system won't be possible. With the rapid advancements in Artificial Intelligence and Machine learning techniques in the past few years, a path has been paved to ease out the generation of Analog Layout Design. And one such open-source tool designed to ease the process is ALIGN (Analog Layout, Intelligently Generated from Netlists).

ALIGN (Analog Layout, Intelligently Generated from Netlists) is an open-source automatic layout generator for analog circuits jointly developed under the DARPA IDEA program by the University of Minnesota, Texas A&M University, and Intel Corporation. The goal of ALIGN is to automatically translate an unannotated (or partially annotated) SPICE netlist of an analog circuit to a GDSII layout. To understand the internal working flow of ALIGN, you can refer to its documentation: ALIGN Documentation

The main objective of this internship is to automate the process of generating the Analog Layout in Cadence Virtuoso using ALIGN. ALIGN expects a netlist to generate the Layout, which would be fed from Virtuoso, a widely used Electronic Design Automation (EDA) tool suite for Integrated Circuit design developed by Cadence Design Systems. Upon processing the netlist and generating the Layout by ALIGN, we would like to import the output GDSII file into our Virtuoso environment as per the PDK used in the Virtuoso. We want to automate the process as mentioned above using Cadence SKILL language, a programming language developed by Cadence Design Systems for use within their Electronic Design Automation (EDA) tools. It is a powerful scripting language specifically designed for automating tasks, customizing workflows, and extending the capabilities of Cadence EDA tools.

Cadence SKILL can execute shell commands, and we would use it to interconnect both tools for our automation. Upon successfully automating the process, we would like to modulate the design generation of the Layout by ALIGN to yield optimized results. We won't be modifying the ALIGN in itself, which might instead create conflicts in the future as the tool evolves with time; instead, we will be using its feature to take in up some constraints as per our requirements.

This internship report describes and explains all the steps to achieve the above-mentioned goals.

Work:

Installation of ALIGN:

Our first task would be to install ALIGN on our Linux server. The general process to install ALIGN is specified in the documentation: ALIGN Installation. But you may encounter problems if some essential packages are missing or have become obsolete in your Linux server. And in some cases, it can be challenging to install the packages without sudo privileges.

But there's an easy workaround to this situation. And that is to install Miniconda on your Linux server, which is a free minimal installer for conda. It's an open-source package management system and environment management system. The steps to install Miniconda on your Linux server are as follows:

- Head toward the official website of Miniconda.
- Download the latest Linux Miniconda3 installer to your server. It would be a file with a ".sh" extension.
- Open the folder where you have downloaded the file and open the terminal there (you will find the "open in terminal" option when you right-click on the blank area in the current folder).
- Type the following commands in your terminal: (Replace "filename" with the actual name of the downloaded file)
  - chmod +x filename
  - ./filename
- Now proceed to install it as per the installation guide displayed in the terminal. Ensure you install it in a directory under the home location.
- Now you need to add the executable directory path to your PATH variable. Otherwise, the system won't be able to search for the executable file of miniconda3.
- To add the executable directory path to your system PATH variable, run the following command in the terminal: (replace /path/to/directory with the location of bin directory under the Miniconda3 installed directory. For example, in my case, it was /home/intern1/miniconda3/bin/)
  - export PATH="/path/to/directory:$PATH"
- You can check whether the path has been added to the PATH variable by executing the following command: (paths are separated by a colon ':')
  - echo $PATH
- Now you should be able to execute the conda. You can check it by running the following command in the terminal:
  - conda --version

Now on successfully installing Miniconda3, we can proceed to install ALIGN.

Note: Do not exit the terminal session because the path variable is changed only for the current session.

Steps to Install ALIGN:

- Execute the following commands in the terminal to Install ALIGN:
  - cd ~
  - git clone https://github.com/ALIGN-analoglayout/ALIGN-public
  - source /path/to/miniconda/bin/directory/activate (For example, in my case: source miniconda3/bin/activate)
  - cd ALIGN-public
  - python -m venv general
  - source general/bin/activate
  - python -m pip install pip –upgrade
  - pip install -v .

- Installation of ALIGN might take some time. If you have installed ALIGN correctly, you can test it by generating a layout from an example netlist provided by ALIGN. To do so, you can execute the following commands:
  - mkdir work
  - cd work
  - schematic2layout.py ../examples/telescopic_ota -p ../pdks/FinFET14nm_Mock_PDK/

- If you encounter a bug as follows: (This is a bug issue at the time of making this report)

ImportError: libOsiCbc.so.3: cannot open shared object file: No such file or directory. Then execute the following command just before executing schematic2layout.py command in the terminal: (This needs to be done always for new terminal sessions)

- export LD_LIBRARY_PATH=${LD_LIBRARY_PATH}:$HOME/ALIGN-public/general/lib

Automation of Layout Design:

Now, after the successful installation of ALIGN, we can move towards automating the Layout generation by integrating the ALIGN with the Cadence Virtuoso. This integration would be done with the help of the Cadence SKILL language.

I have created a script called "automation_script" to automate the process. This script needs to be placed in the "cadence_project" directory. You can find the script at (may require permission to access): Automation Files

Let us see the contents of the script for better understanding so that future improvements can be encouraged:

The script contains five user-defined functions, each performing specific actions associated with a particular task.

- s2l()
- theNetlistGenerator(lib_name cell_name view_name)
- theFeedOfNetlist(cell_name)
- theStreamImporter(lib_name stream_file layer_file)
- theLayoutOpener(lib_name ncell_name)

Let's discuss each function in detail:

s2l(): (s2l is short for schematic-to-layout)

This is the main function that would be executed at the start of the automation process. This function accepts no input arguments and is the source to execute the other user-defined functions systematically with proper input arguments.

The function starts with fetching out the active library, cell, and view names. The below-mentioned commands are used to achieve the library name, cell name, and view name, respectively, and store them in the specified variables (lib_name, cell_name, and view_name):

- lib_name=geGetEditCellView()~>libName
- cell_name=geGetEditCellView()~>cellName
- view_name=geGetEditCellView()~>viewName

These variables would serve as input arguments to other user-defined functions for proper execution. Starting up with theNetlistGenerator function.

theNetlistGenerator(lib_name cell_name view_name):

As the name suggests, this function generates the netlist, which would serve as the input to ALIGN for the Layout generation. The function starts with defining the simulator type used to generate the netlist (we want spice as per ALIGN requirements). The simulator type is set to hspice with the help of the following command:

- simulator('hspiceD)

Then we use the design function to specify the design to which we want to generate the netlist. The function takes in three arguments: library name, cell name, and view name, which is provided by the above-mentioned design variables (lib_name, cell_name, and view_name). The following command is used to achieve this:

- design(lib_name cell_name view_name)

Now we are ready to generate the netlist, and this is achieved by executing the command:

- createNetlist(?recreateAll t)

Now we proceed to our next user-defined function in s2l: theFeedOfNetlist

theFeedOfNetlist(cell_name):

This function takes in cell_name as its argument. As the function name suggests, it feeds the generated netlist to ALIGN. But before providing the netlist directly to ALIGN, we need to make sure that it is compatible to be read by ALIGN. Hence, we have to make some modifications to the generated netlist file before feeding it. These modifications would also depend on the PDK being used in Virtuoso. We will make these modifications and feed the modified netlist to ALIGN using Bash. SKILL language provides an in-built sh() function to run shell commands as child processes. But the drawback is that it does not transmit the defined bash variables from one shell child process to another. Hence, we must run all the commands involving variables under one child process.

Before modifying the generated netlist, we need to access it, and to do so; we need to fetch our cell_name variable into a bash variable. So, our first task under this function would be to copy the value of the cell_name variable to a bash variable. To do so, first, we generate a text file "n.txt," which stores the value of the cell_name variable through SKILL and then fetch out the value using Bash and store it in a bash variable.

First, to write a text file, we need to create an output port in SKILL to open a text file that can be modified. To do so, we use the command:

- n=outfile("n.txt")

This command would create and open a text file called "n.txt" in the default "cadence_project" location. The container of this port is designated as 'n,' which could be used to access this port. Next, we would like to write the value of cell_name to this file, and to do so, we use the following command:

- fprintf(n "%s" cell_name)

This fprintf function takes in the first argument as the output port, which in our case is 'n,' and then the format specifier, in our case its string, hence "%s," and then the value of the string, which is the cell_name. As soon as this command is executed, it will write the value of cell_name to the text file. But to save it and close the port, we need to use the command:

- close(n)

Now we would create a bash variable to store the value of the text file "n.txt," and to do so; we execute the following command as an argument to the sh() function:

- cellname=$(cat n.txt);

Now we will delete the text file since we no longer require it. And to do so, we execute the following command: (Note: This and further specified commands until explicitly mentioned would be inside the same sh() function. Using another sh() function would lead to loss of the defined bash variable.)

- rm n.txt;

Now we have to create a source folder in our home directory. This step is not a part of automation because it is a one-time creation. So, make sure to create a source folder in your home directory. You can do it by executing the following command:

- mkdir $HOME/source

Now we will define a bash variable, which would store the username. This variable is being created for proper navigation purposes. The bash variable would be called "user," and to store the username, execute the following command: (whoami is a bash command which prints out the username)

- user=$(whoami);

Now we will copy the generated netlist from its original location to the source folder; so that we can modify it. To copy it, we use the command:

- cp /data/$user/simulation/$cellname/hspiceD/schematic/netlist/netlist $HOME/source/netlist;

In the above command, we see the use of our defined bash variables, "cellname" and "user." Here it has been used to specify the netlist file's location correctly.

Now we change our working directory to the source directory by using the following command:

- cd $HOME/source;

Now we will execute another Bash script file called BashCode, which will provide the required modifications to this netlist file to ensure it is compatible with ALIGN. You may need to edit this file depending on the pdk used in Virtuoso. This Bash script file can be found at the following host (may require permission to access): Automation Files

Save this Bash script file in the source directory, and then make sure to execute this command in the terminal:

- chmod +x BashCode

This command will not be part of automation as it needs to be only executed once. This command provides sufficient permissions to run the Bash script as an executable file. Now we can run the Bash script by the following command:

- ./BashCode;

The modifications provided by the following script are discussed in a later section. This bash script takes the netlist file as an input and outputs another netlist file called "output." Now we no longer require the original netlist file. Hence we can delete it by the following command:

- rm netlist;

The generated output file is our modified netlist file compatible with ALIGN. But to feed it to ALIGN, first, we must ensure that it is appropriately named since ALIGN searches for the ".sp" extension file as an input. Also, the name of the file should correspond to the cell name. And sometimes, we may require some constraints along with the netlist file to be used by ALIGN, and for that purpose, we would need a separate folder for each type of netlist. This folder can be created by executing the following command:

- mkdir $HOME/source/$cellname;

This command creates a folder in the source directory with the value of the "cellname" variable. Now we would rename our file to the proper format and place it in this recently created folder. To do this, we execute the following commands:

- ext=\".sp\";
- name=\"$cellname$ext\";
- mv $HOME/source/output $HOME/source/$cellname/$name;

The "ext" variable is simply a string containing the ".sp" extension, which is used to concatenate with the "cellname" variable and store it in another variable called "name." Now with the help of the move command, we are moving the output file to the proper location as well as renaming it. Now the use of \" in defining the "ext" and "name" variables is because these commands are input to the sh() function, and this function accepts the input between the double quotes. So, to specify the SKILL to ignore this particular double quote as an ending quote to the sh() function, we use a backslash before it.

Now we can source our created "general" Python environment and run the ALIGN tool for the modified netlist file. We will use a PDK provided by ALIGN called "Bulk65nm_Mock_PDK" as the PDK input to our schematic2layout.py script. This is done by executing the following commands:

- source $HOME/ALIGN-public/general/bin/activate;
- schematic2layout.py $HOME/source/$cellname/ -p $HOME/ALIGN-public/pdks/Bulk65nm_Mock_PDK/

The input to the sh() function ends here. And the role of the theFeedOfNetlist function ends here. At this point, we would have a corresponding GDSII file in our

source folder if ALIGN could successfully generate the Layout without any fatal error. Now we need to import this GDSII file to our Virtuoso. But we may be unable to import the GDSII file to Virtuoso directly. This is because the layer mapping is according to the Bulk65 PDK of ALIGN and may not correspond with the layers used in our particular PDK in Virtuoso. Importing the GDSII file directly may lead to improper layer mapping.

Before using our next function to import the GDSII file, we need to specify proper arguments. So, we created a new variable under the s2l() function called "ncell_name." This variable is being created to correctly identify the GDSII file as per its name. This ncell_name variable is nothing but the capitalized format of cell_name. This variable is created by executing the following command:

- ncell_name=upperCase(cell_name)

Now we will also create a variable called stream_file to store the path of the GDSII file. But you cannot access the "HOME" variable directly in Virtuoso, and for this purpose, we need to create a text file containing the value of the "HOME" variable. Then we need to import the text file, read it, and assign its value to a new SKILL variable. The following command does this:

- sh("echo $HOME > read.txt")
- sh("echo -n $(tr -d \"\n\" < read.txt) > read.txt")
- r=infile("read.txt")
- gets(home r)
- close(r)
- stream_file=strcat(home "/source/" ncell_name "_0.gds")

The above commands first create a "read.txt" file which contains the value of the "HOME" variable. Then the "read.txt" file is modified by removing the newline character using the "tr" command. Then this file is set to an input port "r" in the Virtuoso environment, and the "gets" function is used to get the next line specified from the input port. The "gets" function accepts its first argument as the variable to store the line and, second, the port, which needs to be read for the next line. Then we close the input port by the "close" function. Now we have enough data to specify the path of our stream file. The SKILL variable called stream_file will be created to store the location of the GDSII file. The file's path is achieved by concatenating the string values of the variables and other required strings by the in-built SKILL function called strcat().

Similarly, we would also like to define the path of our layer map file so that our function could access it. The following command does this:

- layer_file=strcat(home "/source/file.layermap")

Now we can move on to our next user-defined function called theStreamImporter.

theStreamImporter(lib_name stream_file layer_file):

This function accepts two arguments; one is the library's name, and the other is the GDSII file's path. Hence lib_name goes as its first argument and stream_file as its second argument. Now we will use the Xstream functions to stream in the GDSII files. The commands executing this task are as follows:

- xstInSetField("strmFile" stream_file)
- xstInSetField("library" lib_name)
- xstInSetField("layerMap" layer_file)
- xstInDoTranslate()

The first command (from the above list of commands) defines the file to be streamed in. The second command defines the library to which the file would be streamed. The third command defines the layer map file that must be used to stream in the file. The fourth command is finally used to stream in the file as per the parameters set by the first three commands.

Now on successfully importing the Layout, we would like to open it. Our next user-defined function would help us in this task, i.e., theLayoutOpener.

theLayoutOpener(lib_name ncell_name):

This function takes up two arguments: the library and the cell name (Here, we would be using the "ncell_name" variable as our second argument). The task of this function is to open the streamed in Layout. And it does this by executing the following commands:

- full_name = strcat(ncell_name "_0")
- geOpen(?lib lib_name ?cell full_name ?view "layout" ?mode "a")

The first command from the above two is used to define a variable called "full_name," which stores the value of the cell name of the layout file. Next, an in-built Virtuoso function "geOpen," is used to open the specified design in the new window. In this case, this function takes in three arguments: the library name, the cell name, and the mode in which it must be opened. Here the mode 'a' stands for edit mode. If you want it to open in read mode, you can change the value to 'r.'

At this point, we have completely understood the main automation_script, and now we will move on to understand the BashCode and Layer Map files.

Understanding the BashCode file:

The BashCode file is a Bash script file used to modify the original netlist file to a format compatible with ALIGN. This Bash script takes the original netlist file as an input and produces an output file called "output." Now ALIGN takes information from the lines between and containing the keywords ".subckt" and ".ends." So, we would like

to delete unnecessary lines in the netlist and only extract the lines between and containing the ".subckt" and ".ends" keywords. Hence at the start, we define four bash variables as per the following commands:

- start_keyword=".subckt"
- end_keyword=".ends"
- input_file="netlist"
- output_file="output"

Now we define a Boolean variable called "extracting," which would be used to specify the lines to be modified. Initially, it is set to false. The command for this operation is:

- extracting=false

Next, we run a while loop which reads each and every line of the original file:

- while IFS= read -r line; do

Here the value of IFS, which is a field separator variable, is set to read the whole line as a single string. Next comes our first if statement, which checks whether the line contains the start keyword. If yes, it sets the "extracting" value to be true and prints out that particular line to the output file. Here are the commands associated with this operation:

- if [[$line == *"$start_keyword"*]]; then
- extracting=true
- echo "$line" >> "$output_file"

The next is an elif statement which checks whether the line contains the end keyword. If it contains, it sets the "extracting" value to false and prints out that particular line to the output file. Commands related to this are as follows:

- elif [[$line == *"$end_keyword"*]]; then
- extracting=false
- echo "$line" >> "$output_file"

Next is another elif statement that modifies the lines between the start and end keywords. This elif statement checks whether the value of "extracting" is true. If it's true, then it performs the operations specified under it. The commands associated with this particular task are:

- elif [[$extracting == true]]; then
- line=${line//nf=1/nf=2}
- line=${line//nmos/n}
- line=${line//pmos/p}
- echo "$line" >> "$output_file"

Here, the line variable is modified in the above commands and is finally printed out to the output file. According to the above commands, if the string contains "nf=1," it is replaced with "nf=2," as ALIGN takes only even values of "number of fingers." Then if the line contains "nmos," it would be replaced with "n," and if it encounters "pmos," it would be replaced with "p," which is a replacement of the name of the MOSFET as per ALIGN devices. Note: You may need to add or modify the above modifications based on the PDK you use in Virtuoso.

Then the "if" statement ends with the "fi" command, and the while loop ends with the "done" command. The redirection (<) is used to specify to read the netlist file as an input to the while loop.

Understanding the layer map file:

A custom layer map file may be required, as the GdsLayerNo and GdsDataType of the PDK you use may not match with those of Bulk65 PDK. Hence, to correctly map the layers defined in the GDSII file in Virtuoso, we need to provide our layer map file, which corresponds with the Bulk65 PDK. Hence, this custom layer map file provides the corresponding GdsLayerNo and GdsDataType according to the ALIGN Bulk65 PDK layer definition.

Loading our automation script file into Virtuoso:

We need to load our automation script each time we initialize a new Virtuoso session. Otherwise, the user-defined functions wouldn't be loaded, and the automation would fail. We can specify Virtuoso to always load the automation script by adding the following command in the ".cdsinit" file, which is present in the "cadence_project" directory (If you are unable to find the file, then make sure you have the option to show hidden files enabled):

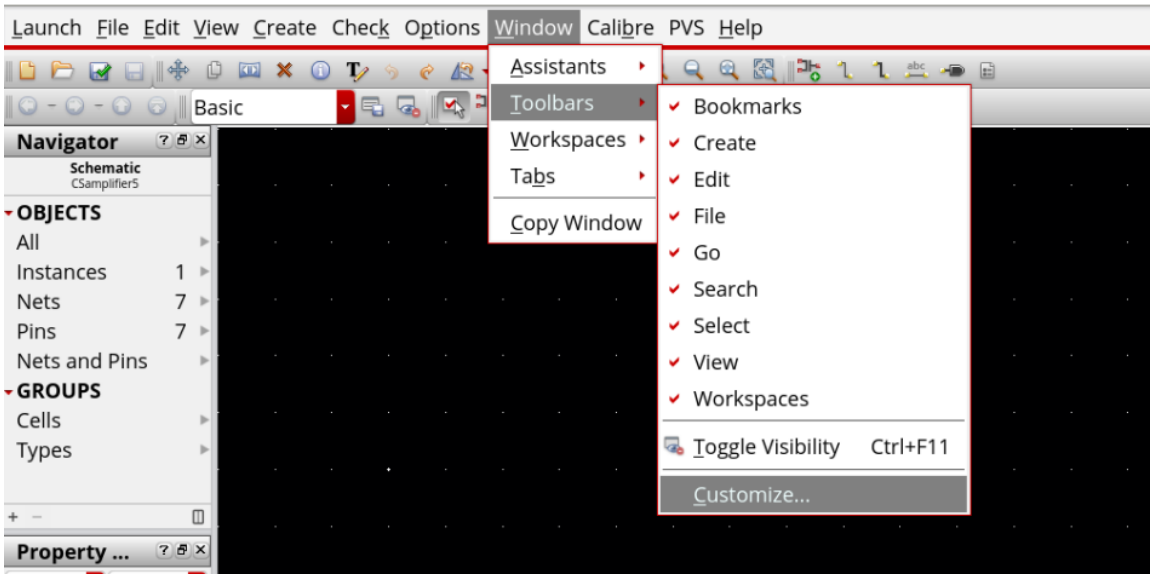
- load(“automation_script”)

This ".cdsinit" file contains the commands which are to be run each time a Virtuoso session is initialized.

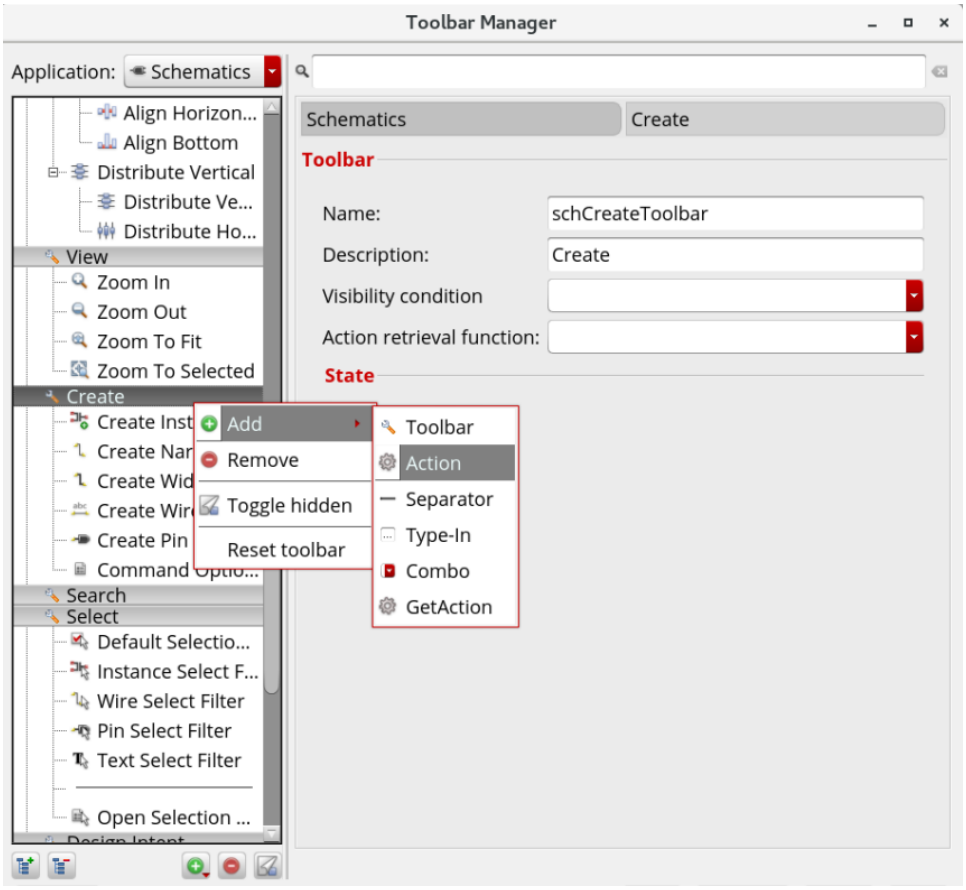
Toolbar icon to run the automation script:

Now, we would be adding a toolbar icon that can run the automation script with just a left click of the mouse. To do so, follow the below steps:

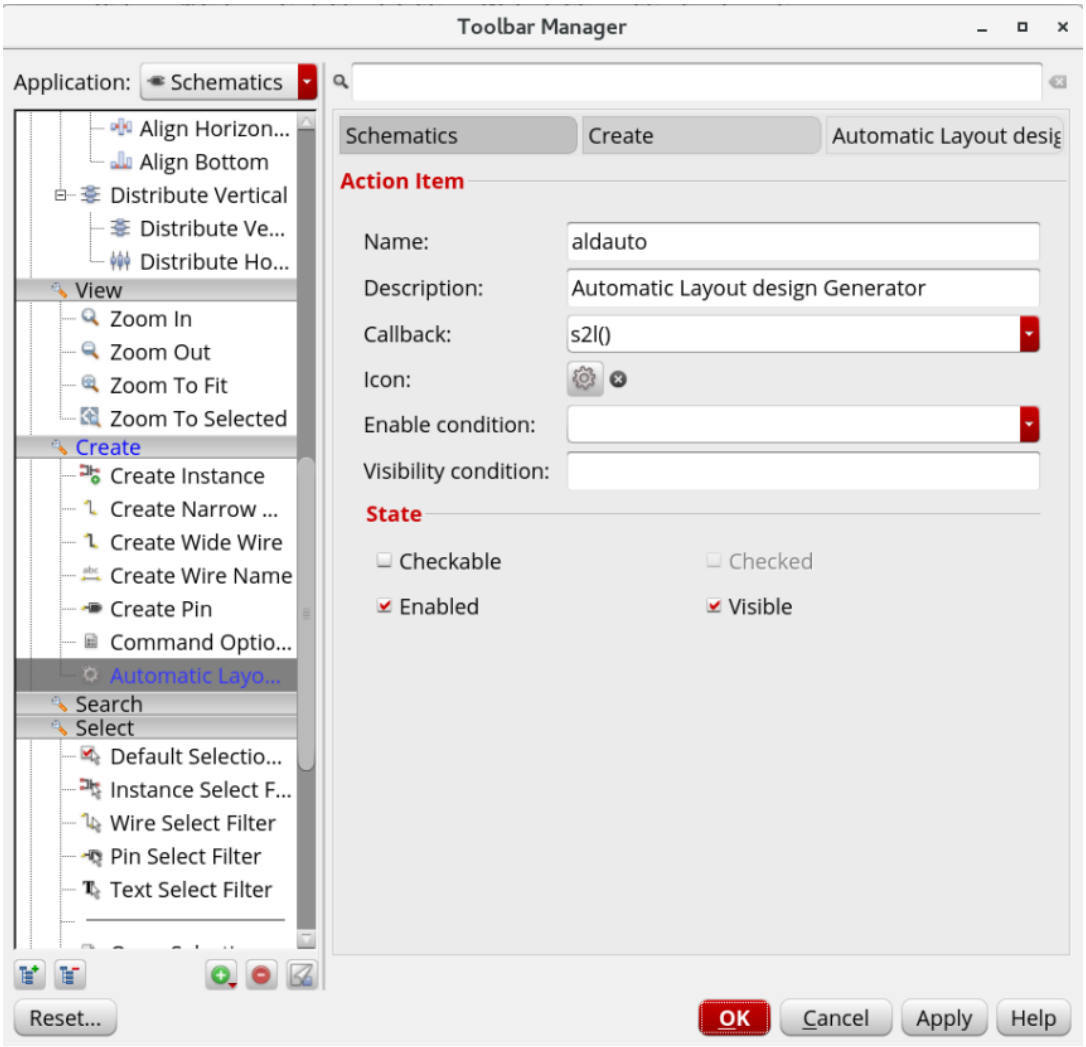
1) Open any schematic in Virtuoso.
2) Then select the "Window" menu, then the "Toolbars" option, and then choose "Customize," as shown below.



3) Now a Toolbar Manager window would pop up. Now right-click on the "Create" section, then choose the "Add" option, and then "Action" as specified in the below image.



4) Now enter the name as "aldauto," the description as "Automatic Layout Design generator," and the callback as "s2l()," also make sure that the "Enabled" option and the "Visible" option are active, as shown below:



5) Now click on "Apply" and then "OK." This finally creates the option of an icon, which can be used to run the automation script.

Constraint Inputs to ALIGN:

ALIGN can take up some constraints to modify the layout design as per the user's needs. The constraint file is optional to generate the Layout. But it would be required in many aspects to direct the Layout generation for optimized results. To know about the constraints that ALIGN can take in, you can refer to the ALIGN documentation: ALIGN Constraints

Some Important points:

- ALIGN reads the content of the netlist file, which is defined between the ".subckt" and ".ends." Hence, you must generate the Layout from a schematic which should define the whole schematic as a sub-circuit.
- To achieve this, you need to generate a cell view of your circuit and then create a new schematic view with the same cell name in another library (as you cannot create another schematic with the same cell name in the same library). And then, you need to add your created cell (aka symbol) in that particular schematic. Assign the required pins and then check and save the schematic.

- If you get no errors and warnings after you check and save it, you can run the automation script by clicking on the "Automatic Layout Design Generator" icon on the "Create" toolbar.
- To provide constraints before layout generation (i.e., automation), you need to create a folder in the source directory, whose name should be the same as the cell name of the schematic. Then you can add your constraint file to this recently created folder.
- Note: The name of the constraint file should be <cellname>.const.json; replace the <cellname> with the cell name of the schematic.

An Example to demonstrate the automation:

Layout Automation of CSamplifier:

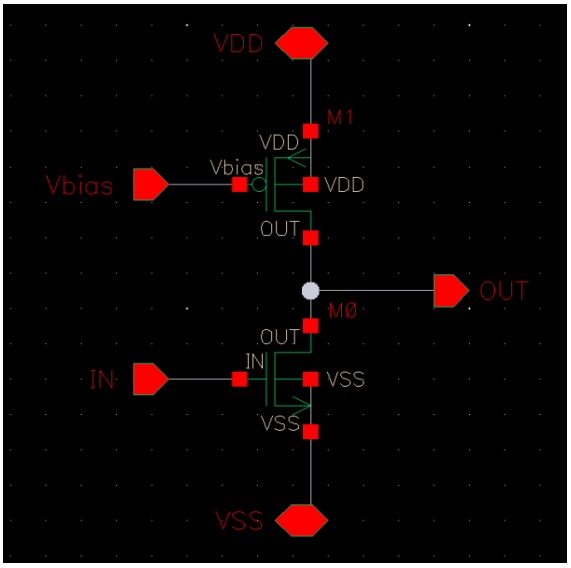
Our first task would be to create the schematic of the Common Source amplifier. And to create it, we would be using two MOSFETs; one would be of p-type, which would function as the current source, and the other would be of n-type as our amplifying transistor. Here, we aim to generate the Layout through automation:

First Schematic:

Library = Example

Cell name = CSamplifier

View name = Schematic



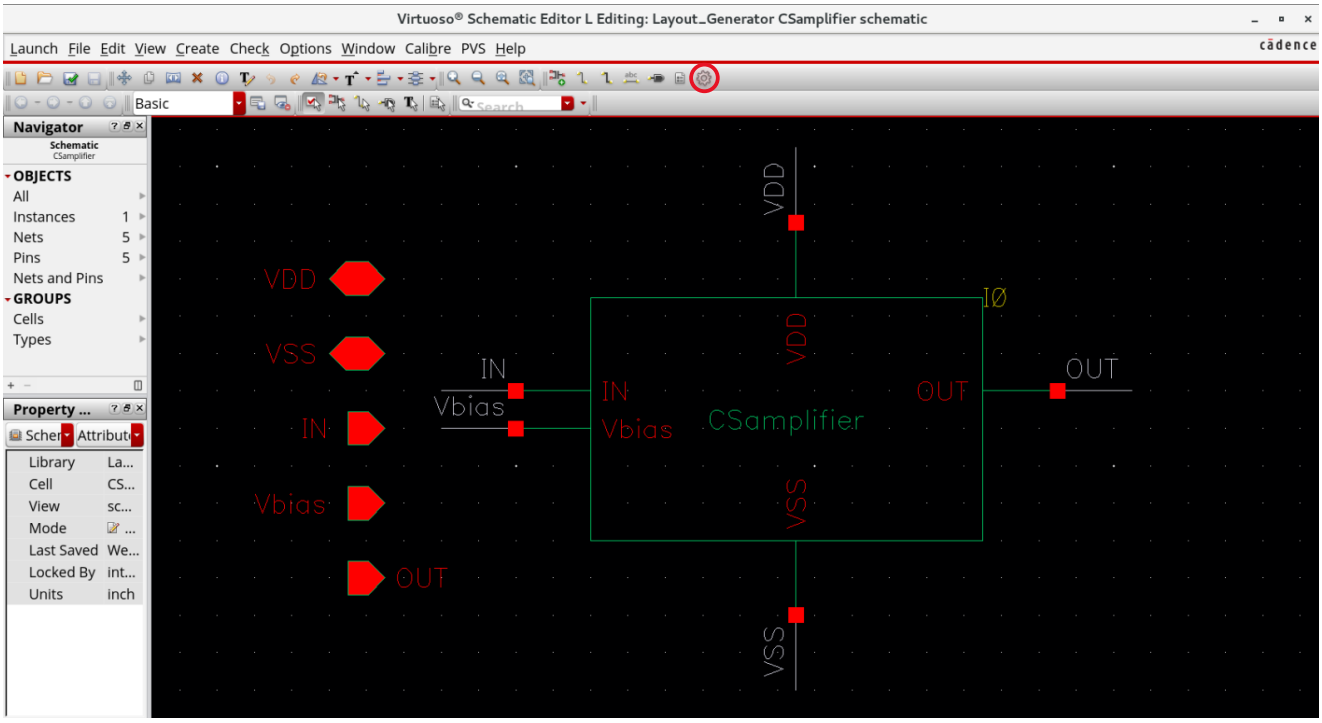
After creating this schematic, we need to create the cell view, which would be used to create another schematic with the same cell name in a different library. This is necessary to generate a proper netlist file compatible with ALIGN.

Second Schematic:

Library = Layout_Generator

Cell name = CSamplifier

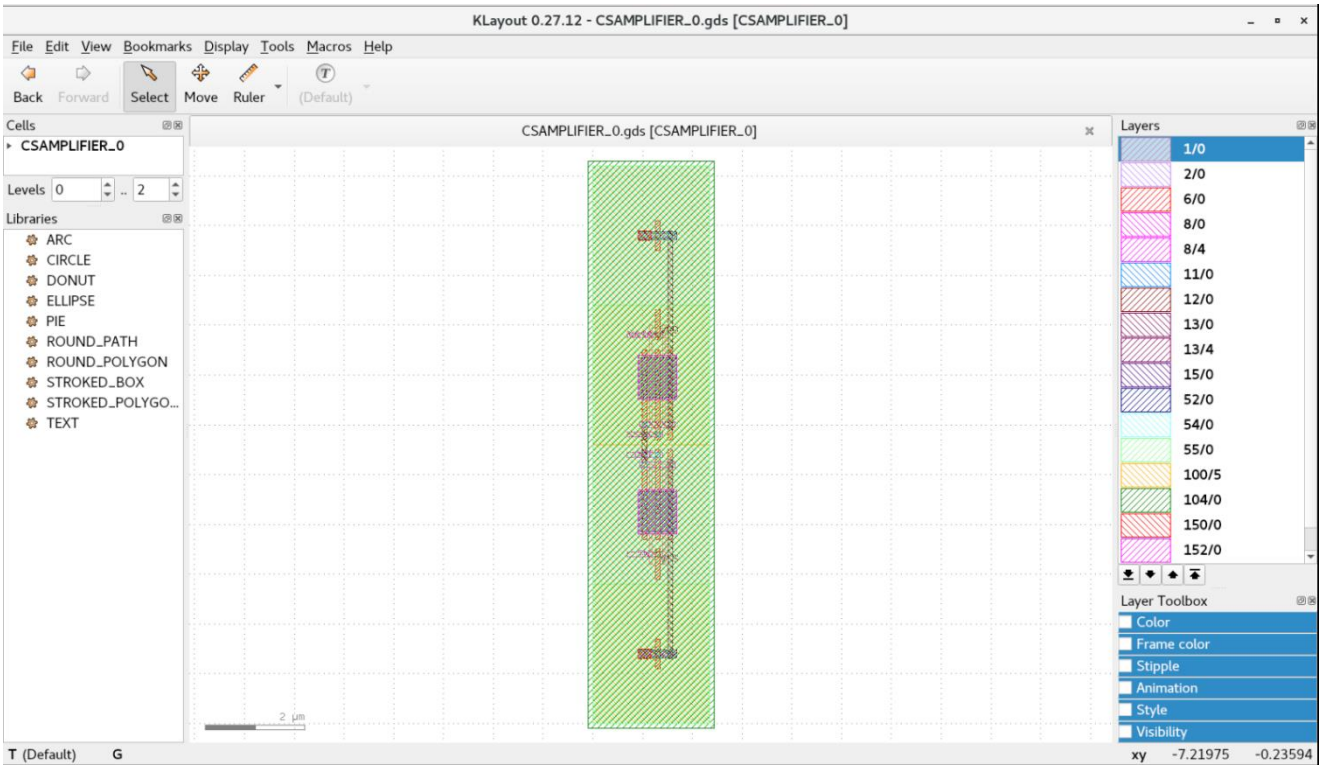
View name = Schematic

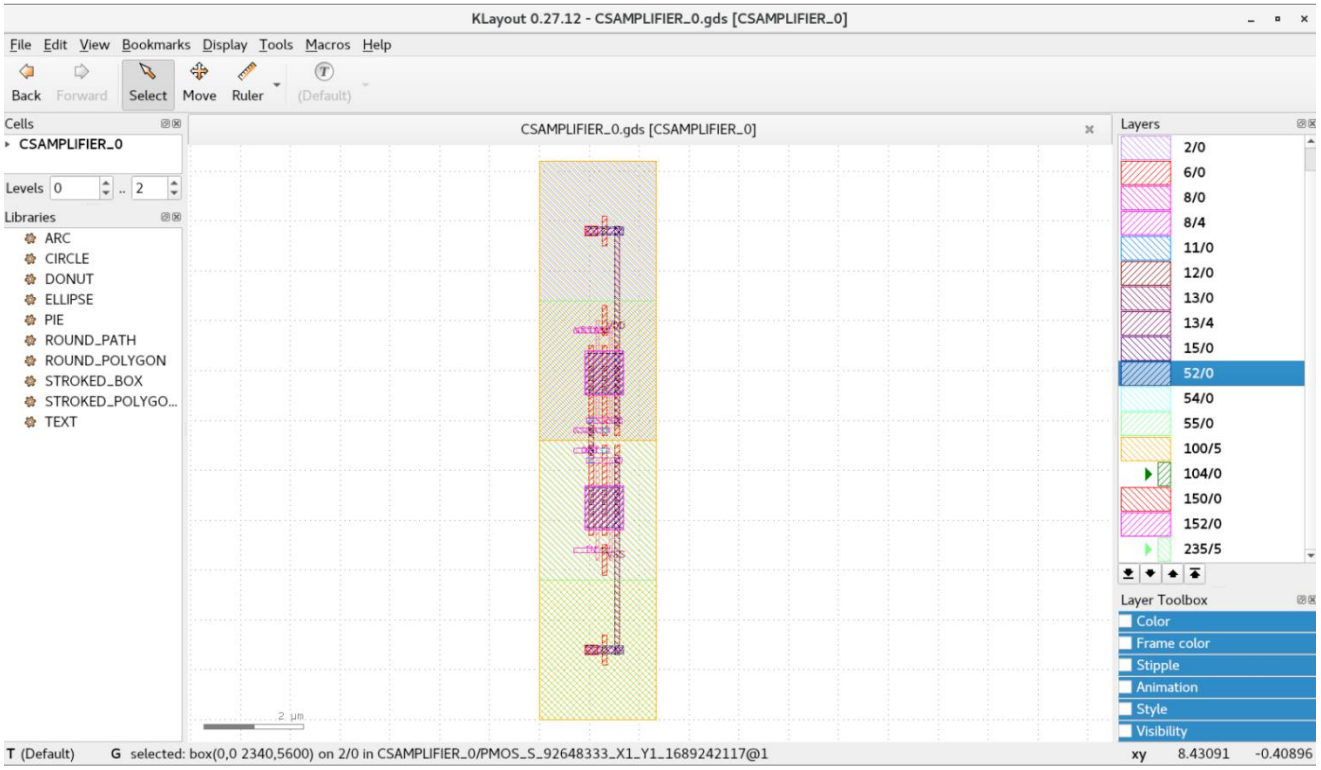


Now, we can run the automation script by clicking the gear icon (circled in red in the above image) we created in the "Create" toolbar. A successful layout generation would not result in any error, and you can check this in the terminal. In our case, the Layout has been successfully generated, as shown below:

```
align.compiler.compiler INFO : Power and ground nets not found. Power grid will not be const
ructed.
align.compiler.compiler INFO : Completed topology identification.
align.pnr.main INFO : Running Place & Route for CSAMPLIFIER
align.pnr.build_pnr_model INFO : Reading contraint json file CSAMPLIFIER.pnr.const.json
align.pnr.build_pnr_model INFO : Reading contraint json file CSAMPLIFIER.pnr.const.json
align.pnr.placer INFO : Starting bottom-up placement on CSAMPLIFIER 0
PnR.placer.SeqPair.SeqPair INFO : Enumerated search
PnR.placer.Placer.PlacementCoreAspectRatio_ILP INFO : Exhausted all permutations of seq pair
s and found 1 placement solution(s)
align.pnr.build_pnr_model INFO : Reading contraint json file CSAMPLIFIER.pnr.const.json
PnR.placer.SeqPair.SeqPair INFO : Enumerated search
align.pnr.router INFO : Starting top_down routing on CSAMPLIFIER 0 restricted to None
PnR.router.Router.RouteWork INFO : GcellGlobalRouter: CSAMPLIFIER
PnR.router.Router.RouteWork INFO : GcellDetailRouter: CSAMPLIFIER
PnR.router.Router.RouteWork INFO : Create power grid: CSAMPLIFIER
PnR.router.Router.RouteWork INFO : Power routing CSAMPLIFIER
align.pnr.main INFO : OUTPUT json at /home/intern1/source/3_pnr/CSAMPLIFIER_0.json
align.pnr.main INFO : OUTPUT gds.json /home/intern1/source/3_pnr/CSAMPLIFIER_0.python.gds.js
on
Use KLayout to visualize the generated GDS: /home/intern1/source/CSAMPLIFIER_0.gds
Use KLayout to visualize the python generated GDS: /home/intern1/source/CSAMPLIFIER_0.python
.gds
```

Now, the Layout should be successfully streamed in, and you must be able to view it. Layout obtained: (Here, the Layout is shown in KLayout, due to some security concerns)





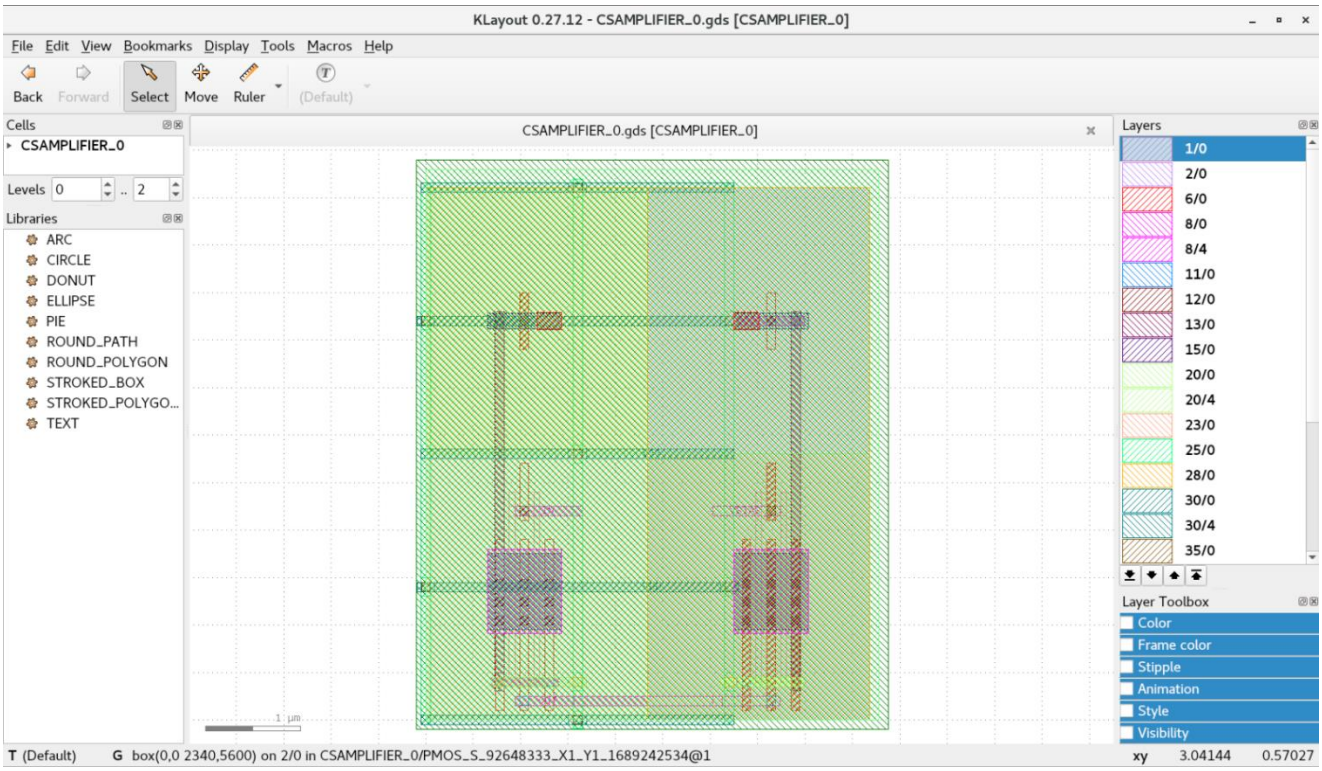
The layout design can be more optimized and tweaked as per your requirements if you provide a proper constraint file to the ALIGN. For example, if you wish that the order of the instances should not be as shown above; instead horizontal, then you can create a constraint file and specify it. But first, you need to create a folder with the cell name in the "source" directory. And then, you have to create a constraint file in it, whose name should be of this format: <cellname>.const.json (replace the <cellname> with your cell name.)

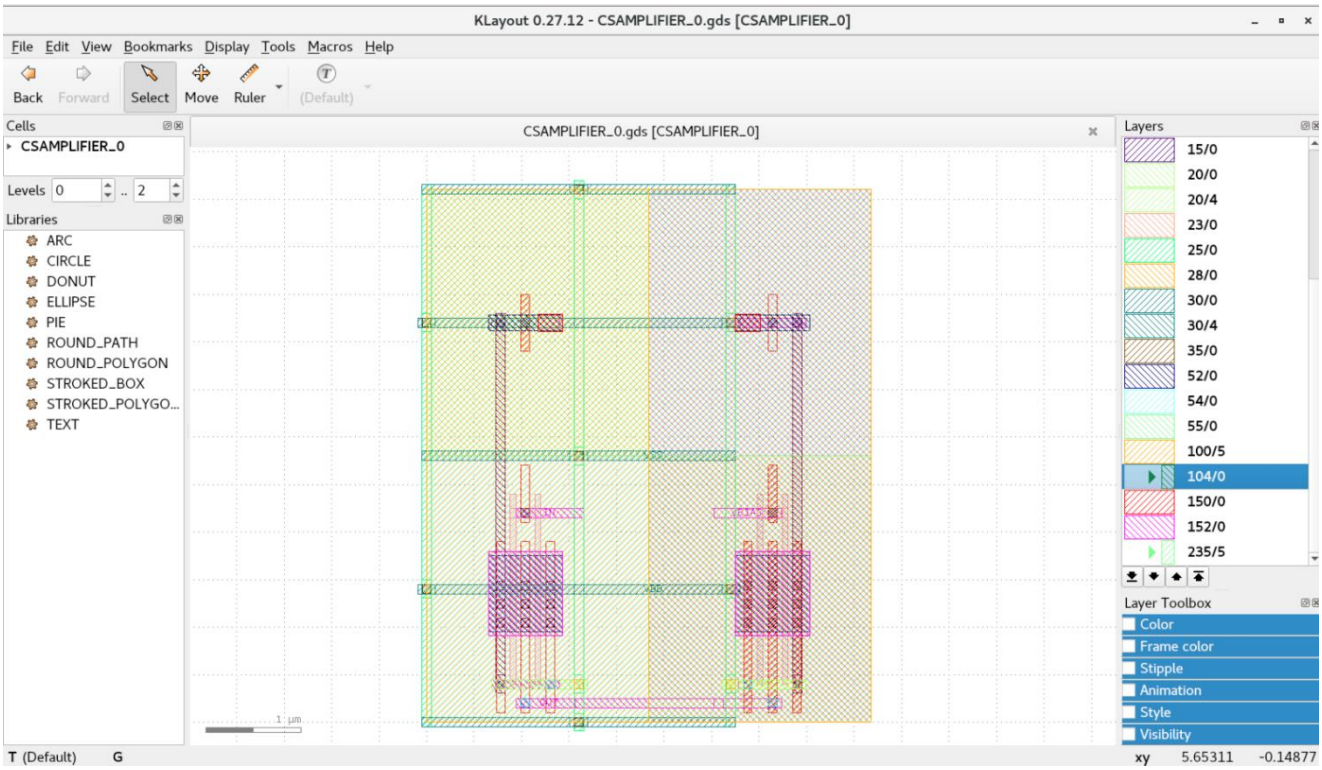
In my case, the cell name is CSamplifier. Therefore, I will create a CSamplifier folder in the "source" directory; and then create a CSamplifier.const.json file in the created folder. As per the customization mentioned above, I would need a constraint file with constraints as shown below:

CSamplifier.const.json
~/source/CSamplifier

```
[
        {"constraint": "PowerPorts", "ports": ["VDD"]},
        {"constraint": "GroundPorts", "ports": ["VSS"]},
        {"constraint":"Order", "instances": ["M0", "M1"], "direction": "horizontal"}
]
```

This first and second constraint tells that "VDD" is the power port and "VSS" is the ground port of the hierarchy. And then, the third constraint specifies that the instances "M0" and "M1", which are our n-channel and p-channel MOSFETs, respectively, must be placed horizontally. The resulting Layout is shown in the below image: (Here you can see that the MOSFET instances are being placed in the horizontal fashion, which may not be desirable from the point of optimization but is shown to demonstrate the example of the use of constraints)





You can also edit the Layout as per your needs and can move further. Here the demonstration of the example ends.

Results:

ALIGN is currently under development and is expected to improve further in the coming days. The Layout produced by ALIGN may not be the best or be adequately optimized, but it is expected to do so in the coming days. Currently, it is capable of generating an acceptable working layout. As the Machine learning techniques would evolve, it would make the ALIGN capable of generating better Layouts with time. You may also encounter some bugs, which the ALIGN team is working to fix. These bugs are expected to be resolved quickly and offer seamless automated layout generation.

Conclusion:

Currently, the automation is in a working state but has a scope for further advancement. For example, as per the requirements of the PDK, few default constraints could automatically be generated, which would further decrease the manual input and save the user's time. As ALIGN continues to evolve, this automation will become increasingly helpful and capable of generating a layout comparable to that of a human standard.