Relational Framework for Prompt-based Automatic Layout Generation with Glayout

Name - Chetanya Goyal
Slack - Chetanya Goyal
Email - chinnidude@gmail.com
GitHub - chetanyagoyal

Aim

Develop a system to generate DRC-clean analog layouts from natural language prompts, streamlining the layout creation process and enabling automated high-level circuit specification.

Background

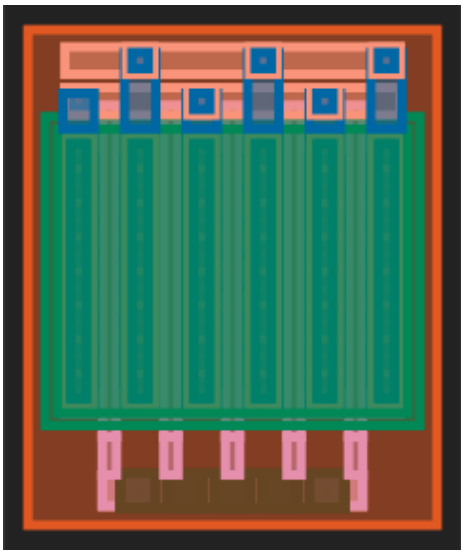
Glayout is an Analog Layout Automation tool that uses a fully hierarchical framework (with a gdsfactory backend) that allows the user to write python code instead of having to manually create layouts in tools such as Magic VLSI. The API uses a "MappedPDK", which consists of the governing set of rules (g_rules) for a certain pdk and the generic set of layers (g_layers). This allows the user to abstract out the actual name of a layer (stored as an integer tuple) for a particular technology (such as sky130 or gf180), as a result of which they can program in a pdk-agnostic environment. The API consists of a set of Pcells (parametric cells), routing methods, component movement methods and transistor placement methods.

The Pcells themselves (implemented as pythonic code) are DRC clean components which can be used to create larger designs, in conjunction with the routing and movement methods. An example of this is the nmos Pcell, which can be found here. This cell can be customized and placed as required to create larger designs. The .gds output can be obtained

by using gdsfactory's write_gds() function. The visualized layout has also been shown below. Glayout already implements larger circuits such as differential pairs, operational amplifiers, current mirrors, etc. using the currently existing primitives.

```python
@cell
def nmos(
    pdk,
    width: float = 3,
    fingers: Optional[int] = 1,
    multipliers: Optional[int] = 1,
    with_tie: bool = True,
    with_dummy: Union[bool, tuple[bool, bool]] =
True,
    # ... other parameters
) -> Component:
```

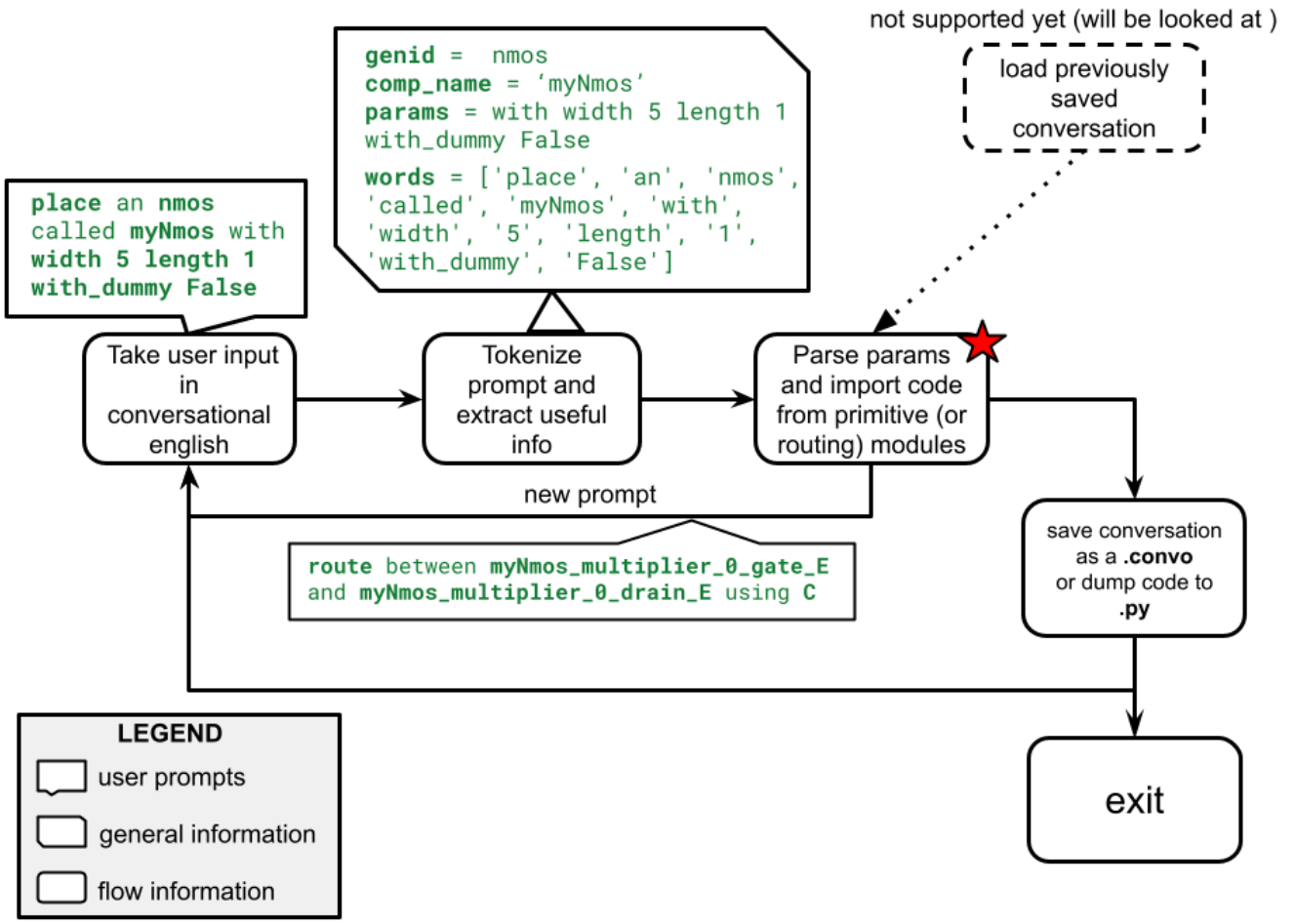
Code for an nfet Pcell (with pfets, op-amps, diff_pairs, etc written similarly in python code)

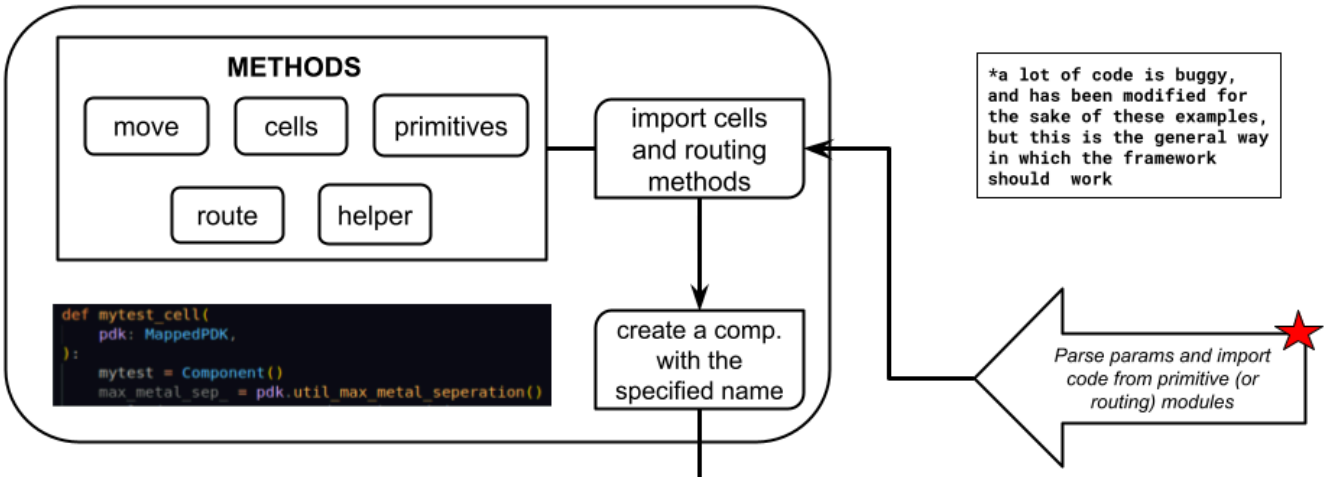


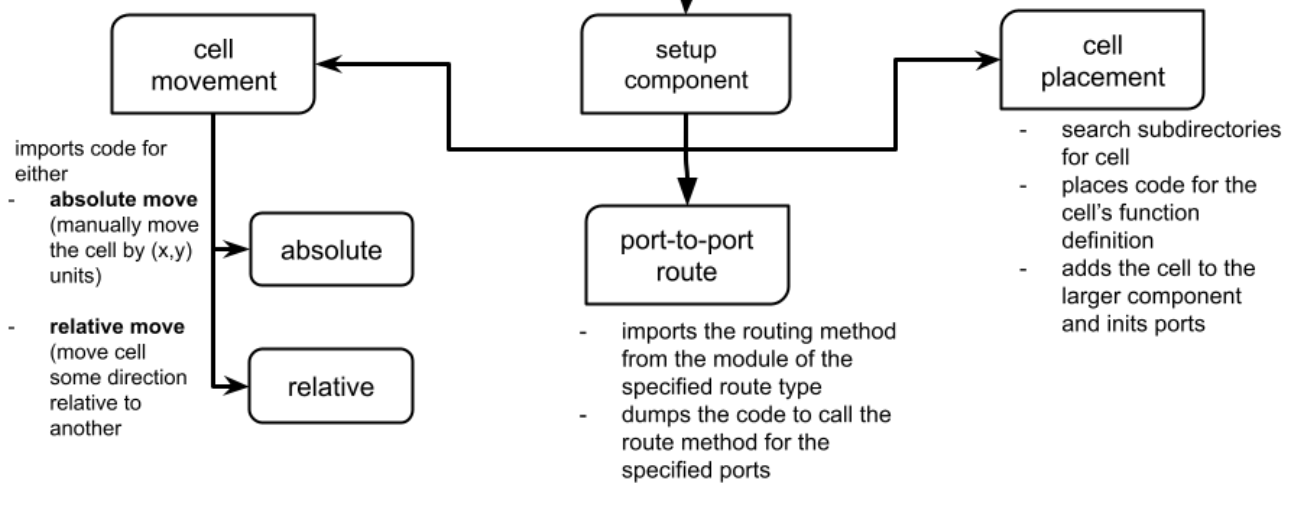
Layout of an nmos with 5 fingers and minimum size

Implementing larger cells (or circuits), especially those as large as (or larger than) Operational Amplifiers, requires a tremendous amount of work, even with this framework. This calls for either a simplification of the codebase or a framework built on top of the API that abstracts out the need for the user to both be a skilled coder and to know about the API's specifics. This project seeks to do the latter, by expanding and improving upon an experimental prompt-based framework (in the 3 top level python files in the folder, originally authored by this contributor).

This framework (in its current state) works in a way that is described by the figures below.
The framework uses nltk's sentence tokenizer to split a sentence into words stored in a list and parses the list to obtain the main function (place, move, route, etc.), the name of the module to be placed, port names if routing and subcomponent references if moving. Conversations consisting of user prompts can be saved as .convo files, which the user will ideally be able to load in on the fly.


not supported yet (will be looked at)
load previously saved conversation
genid = nmos
comp_name = 'myNmos'
params = with width 5 length 1 with_dummy False
words = ['place', 'an', 'nmos', 'called', 'myNmos', 'with', 'width', '5', 'length', '1', 'with_dummy', 'False']
place an nmos called myNmos with width 5 length 1 with_dummy False
Take user input in conversational english
Tokenize prompt and extract useful info
Parse params and import code from primitive (or routing) modules
new prompt
route between myNmos_multiplier_0_gate_E and myNmos_multiplier_0_drain_E using C
save conversation as a .convo or dump code to .py
exit
LEGEND
user prompts
general information
flow information


METHODS
move
cells
primitives
route
helper
import cells and routing methods
*a lot of code is buggy, and has been modified for the sake of these examples, but this is the general way in which the framework should work
def mytest_cell(
pdk: MappedPDK,
):
mytest = Component()
max_metal_sep_ = pdk.util_max_metal_seperation()
create a comp. with the specified name
Parse params and import code from primitive (or routing) modules


cell movement
setup component
cell placement
imports code for either
- absolute move (manually move the cell by (x,y) units)
- relative move (move cell some direction relative to another
absolute
relative
port-to-port route
- imports the routing method from the module of the specified route type
- dumps the code to call the route method for the specified ports
- search subdirectories for cell
- places code for the cell's function definition
- adds the cell to the larger component and inits ports

The framework is currently able to handle placement, routing and movement for cells, saving conversations and dumping code, and the creation of variables (strings, ints, dictionaries, lists, etc.). The routing also requires the user to know the orientations and internal port nomenclature for routing, which is non ideal. Currently, the only way to assess the proper function of a generated design is to manually run a magic DRC check. A better way to check for correctness would be for the framework to be able to also write netlists when dumping code, on which a magic LVS check can be run. This will ensure that the layout created corresponds to the circuit netlist. Another useful improvement would be to allow for multiple functions to be executed from a single prompt. These are proposed to be looked at as feature improvements. The framework needs to also be able to handle general prompts, such as `place a folded cascode stage called myFoldedCascode` (which involves placing components that have not been implemented as Pcells yet). To account for this, the relational database can be supplemented with a lookup table containing information about the components, routing, placement, etc. about classes of general circuits (such as cascodes, amplifiers, comparators, etc.), implemented as a `.json` file like the one shown below

```
{
  "cascode": {
    "name": "nmos_cascode",
    "components": {
      "nfet": ["path_to_nfet"],
      "pfet_load": ["path_to_pfet"]
    },
    "routes_are_between": {
     "nfet": {
        "drain_nfet_bottom": ["source_nfet_top"],,
        "drain_nfet_top": ["drain_pfet_bottom", "via_out"]
        ... and so on
      }
    }
  }
  ... other circuit classes like amplifiers, cascodes, etc
}
```

Additional Pcells and transistor placement methods will also need to be implemented in the course of this project to make the transition to an LLM easier. This would be the first implementation of a natural language to layout tool, and would be a useful alternative to current netlist-to-layout tools as it would significantly reduce the barrier-to-entry for using glayout for analog design and would reduce the complexity of the analog design process in general.

Deliverables

- Implement DRC checks for Glayout using magic DRC in OpenFASoC's CI system
- Refine relational framework with bug fixes and detailed error logging
- Implement more complex Pcells and placement methods (enumerated here)
- Improve the Relational Database:
  - Abstract out internal port names to simplify routing
  - Allow for multiple commands (separated by the word "and") in a single prompt
  - Create lookup `.json` files for generic circuit classes
- Implement automatic netlist generation to enable LVS checks for user created circuits
- Deploy the framework and crowdsource circuit designs to accelerate the generation of a training dataset
- Develop the LLM to simplify prompts into sentences that the relational model can parse and dump code for

Timeline

- **Week 1**: Implement CI checks for Glayout code
- **Weeks 2 to 4**: Implement new Pcells
- **Weeks 5 and 6**: Find and fix bugs in the current relational framework
- **Week 7**: Bolster the error logging of the framework
- **Weeks 8 and 9**: Implement transistor placement methods such as interdigitated, dummy, "ABBA" placement etc.

- **Week 10**: Simplify routing methods
- **Week 11**: Implement multiple commands in a single prompt
- **Week 12**: Create lookup files for circuit classes
- **Week 13**: Implement netlist generation and LVS checking
- **Week 14**: Review feature PR with team
- **Week 15**: Research and discuss with team on optimum method to implement LLM
- **Weeks 16 and 17**: Deploy the framework online and crowdsource circuit designs with their prompts to accelerate dataset creation
- **Weeks 18 and 19**: Pick the most optimum designs and prompts from the dataset, clean and preprocess the data
- **Weeks 20 and 21:** Use an encoder-decoder architecture and pre-train the LLM
- **Week 22 -** Evaluate the model and fine-tune until the desired similarity is achieved between ideal relational prompt and generated relational prompt

Project Length - Large

Mentor - `mehdi@umich.edu`

About Me

I am Chetanya Goyal, a 3rd year undergraduate student at IIIT Hyderabad, India. I am currently pursuing a dual-degree in Electronics and Communications Engineering (B.Tech. + M.S. by Research). I have been looking to apply for GSoC for a while now as it unifies the two areas that I am most interested in - Hardware Design and open-source contribution.

I have worked with Python, NGspice, LTSpice, iVerilog, C++ and Magic extensively over the past 2 years in various course projects and other undertakings. I am relatively new to open-source, having only started contributing to OpenFASoC half a year ago.
Through these contributions, which involve infrastructure improvements (expanding CI checking, decreasing code redundancy, ubuntu22 support) and bug fixes (build file, generally fixing Issues, etc. raised here), I have become well acquainted with the codebase and the ethos.

I have started working on Glayout and can see the potential in the idea, and would really like to be a part of the program, as it would contribute significantly to the EDA community at large.