Human Language to Analog Layout Using GLayout Layout Automation Framework

Ali Hammoud
University of Michigan

Chetanya Goyal
IIIT Hyderabad

Sakib Pathen
University of Michigan

Arlene Dai
University of Michigan

Anhang Li
University of Michigan

Gregory Kielian
Google AI

Mehdi Saligane
University of Michigan

ABSTRACT

Current approaches to Analog Layout Automation apply ML techniques such as Graph Convolutional Neural Networks (GCN) to translate netlist to layout. While these ML approaches have proven to be effective, they lack the powerful reasoning capabilities, an intuitive human interface, and standard evaluation benchmarks that have been improving at a rapid development pace in Large Language Models (LLMs). The GLayout framework introduced in this work translates analog layout into an expressive, technology generic, compact text representation. Then, an LLM is taught to understand analog layout through fine-tuning and in-context learning using Retrieval Augmented Generation (RAG). The LLM is able to successfully layout unseen circuits based on new information provided in-context. We train 3.8, 7, and 22 Billion parameter quantized LLMs on a dataset of less than 50 unique circuits, and text documents providing layout knowledge. The 22B parameter model is tuned in 2 hours on a single NVIDIA A100 GPU. The open-source evaluation set is proposed as an automation benchmark for LLM layout automation tasks, and ranges from 2-transistor circuits to a $\Delta\Sigma$ ADC. The 22B model completes 70% of the tasks in the evaluation set, and passes DRC and LVS verification on 44% of evaluations with verified correct blocks up to 4 transistors in size.

KEYWORDS

Analog Layout Automation, Open Source, GLayout, Retrieval Augmented Generation (RAG), Parameter Efficient Fine Tuning, Large Language Model, Quantized Low Rank Adaptation (QLORA)

ACM Reference Format:
Ali Hammoud, Chetanya Goyal, Sakib Pathen, Arlene Dai, Anhang Li, Gregory Kielian, and Mehdi Saligane. 2024. Human Language to Analog Layout Using GLayout Layout Automation Framework. In *2024 ACM/IEEE International Symposium on Machine Learning for CAD (MLCAD '24), September 9–11, 2024, Salt Lake City, UT, USA.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3670474.3685971

MLCAD '24, September 9–11, 2024, Salt Lake City, UT, USA

ACM ISBN 979-8-4007-0699-8/24/09…$15.00
https://doi.org/10.1145/3670474.3685971

1 INTRODUCTION

The increasing need for Analog and Mixed Signal (AMS) design automation has been studied in recent works [3]. Previous Analog generation tools have used approaches which are optimized for a single type of circuit [9, 20]. Using these tools it is possible to automate the construction of large blocks, but these design generators are specially built for a particular circuit, and must be rebuilt for new circuits. These circuit specific approaches, while providing great performance, are not realistic for building a general Analog design generator. Machine Learning has emerged as an effective approach to the general layout automation problem. Tools such as [12, 21] use Machine Learning techniques such as Graph Convolutional Neural Networks (GCN) to generate layout constraints and translate netlist to layout with good results on unseen schematics.

While these approaches have delivered good results [16], they lack the reasoning capabilities and unified comparison metrics inherent in modern Large Language Models (LLM). The LLM approach can be directly scaled (increasing model and dataset size improves performance 3) and would enable faster development in Analog automation, it also allows for performance to be measured on standard evaluation sets across LLM implementations. For example, LLMs are benchmarked on datasets such as MBPP [4] which evaluates code generation, or PIQA which evaluates reasoning [6]. These evaluation scores provide single number, quantified comparisons between LLMs. If LLM analog automation tools were created, they could adopt a standard evaluation set as common comparisons, which would help unify development efforts.

Additionally, an LLM eliminates the use of constraint files common with ML layout automation tools [16, 21], in favor of abstract human language requests: leveraging the LLMs reasoning ability to fulfil the request, or allowing the LLM to decide automatically if limited information is provided. For example, the LLM can be

Table 1: Summary of LLM design approaches. Glayout uses open-source 3, 7, and 22 Billion parameter LLMs. Previous analog LLMs are not layout capable.

Approach	Design Type	Layout	Open Source	Model Size
[7]	Digital	yes	no	1700 B
[13]	Analog	no	no	1700 B
[14]	Analog	no	no	1700 B
[18]	Digital	yes	yes	16 B
Glayout	**Analog**	**yes**	**yes**	**3, 7, 22 B**

prompted to layout an existing circuit using new styles or layout a new circuit in a known style without being previously trained on this task. For the task in question, existing approaches have no automation mechanisms and would require a manual constraints file rewrite. Furthermore, virtually any proposed task can be automated out-of-the-box with an LLM approach, because of the "AI" assistant interface.

In LLMs, the human language interface also provides ease of use advantages for the human designer. An assistant is more intuitive to work with than a traditional tool which may require previous training. As summarized in table 1, the LLM approach has been applied to digital layout with tools utilizing LLMs to generate Verilog code from a high level digital design request [15]. Verilog generators have been successful at producing valid Verilog with models as small as 16 Billion parameters finetuned in 15 GPU Hours [18]. There have also been tools enabling LLM based Analog schematic design such as [13, 14], but as far as we are aware, there are no existing LLM based systems for Analog Layout.

This work proposes GLayout, an Analog Layout Python API using a LLM to translate general human language user prompts into analog layout. GLayout is entirely open-source and available for public installation via PyPI. GLayout and the corresponding evaluation set can serve as a starting point for future development into Analog Layout capable LLMs.

2 APPROACH

It is not wise to describe raw layouts to an LLM, as this would require many billions of tokens in training data and a massive amount of compute resources to learn useful patterns. Instead, a more optimized approach can be created by considering why digital design is so readily amenable to large language model automation. Digital design can be captured in a highly compressed text based format as a Verilog file. In this case, the raw layout is mostly noisy data and would result in the LLM learning many non-meaningful

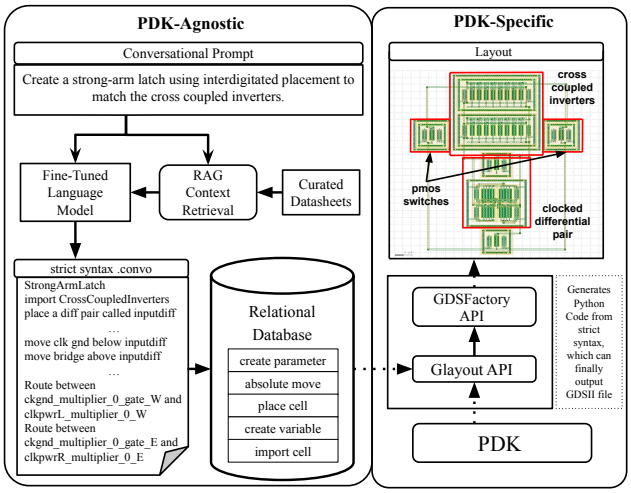


Figure 1: Full process of translating user prompt to a final layout.

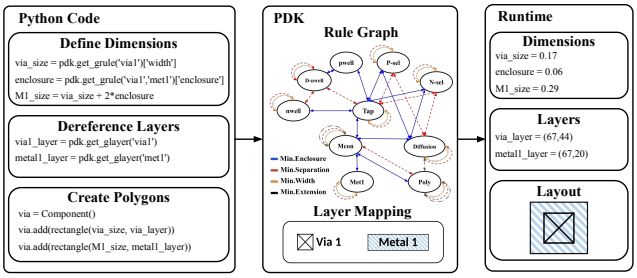


Figure 2: Instantiating a simple VIA written in Python. The process design kit is a parameter.

patterns. Verilog distills a digital design to the description of design intent, which is RTL logic, and omits layout geometry information.

Analog layout can also be compressed to omit noisy data, but it requires a new description format to capture analog layout design. This description should capture the layout topology while omitting layout details. The key to implementing very fast and efficient LLM learning on analog layouts is to distill the layout information down to a simple description. For this, we designed a new command language description format for analog layout topology which we call "strict syntax".

LLMs are already powerful reasoning engines [19] and can be guided to understand analog design (as previously demonstrated) [14]; in-context data can be used to provide layout knowledge and the fine-tuning step teaches the model to express layout using the proposed description format. Furthermore, reasoning performance scales with model size [8] which makes the translation performance directly scalable with larger model size (see 3 for results).

As illustrated in Fig. 1, the steps of our proposed approach from user prompt to final layout are as follows:

(1) The user prompt is passed to the LLM which outputs a strict syntax command file.
(2) The strict syntax file is compiled to a Python function.
(3) The Python function is called, with the PDK (process design kit) and other parameters passed at run time, to output the final layout.

2.1 Python API

The fundamental GLayout engine is Python based and calls the GDSFactory tool [2] for layout manipulation. Circuit blocks written with the API are Python functions which accept several parameters, like normal Parameterized Cells (Pcells), but also accept the PDK as a parameter. A simple example is that of the primitive Pcells. The GLayout transistor primitives accept parameters for width, length, multipliers, (among 15 other parameters), and additionally accept the PDK as one of the parameters.

The PDK is passed in a python class called "MappedPDK", which acts as an interface between specific technologies and the generic GLayout API. MappedPDK maps process specific rules and layers to process-agnostic labels, which enables complete reusability of GLayout Python generator code across different technologies. The MappedPDK stores rules and layers as illustrated in Fig. 2

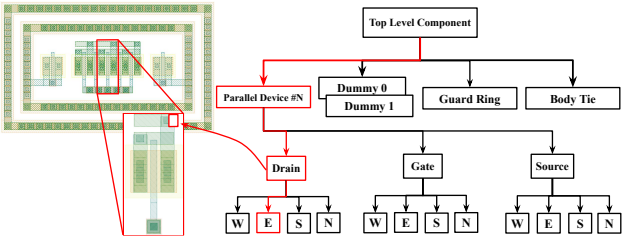


Figure 3: Referencing the drain of an n-type transistor. Ports can be easily referenced thanks to an organized naming standard.

- *Layers*: Different processes use similar layers from the designers perspective (such as "active/diffusion", "metal", "via", etc.) but with different identifiers, represented as integer pairs in the final layout file. For example, 130nm layouts may store the tuples $(67, 44)$ for the via1 layer, $(68, 20)$ for metal1, and $(65, 20)$ to denote active region, while 180nm may store completely different layer identifiers. Manual layout requires an engineer (or design tool) creating designs to know the specific layer identifiers, or to use graphical layout editors to select layers. MappedPDK abstracts PDK specific design layers by mapping layers with different names but similar functions to common identifiers. For example, the identifier "active" is used in the GLayout API to identify the active region, regardless of the underlying PDK. The layer identifier lookup is performed by calling the PDK.get_glayer method.
- *Rules*: Design rules decks used in DRC (Design Rule Checking) consist of rules such as "*min_separation*", "*min_enclosure*" and "*min_width*" which exist between layers. Regardless of technology, combinations of these rules exist between layers. MappedPDK provides a standard way of searching for rules between layers, regardless of the underlying PDK. The rule value lookup is performed by calling the PDK.get_grule method.

In addition to MappedPDK, GLayout provides a library of parameterized-cells (pcells). This includes basic cells such as transistors, capacitors, and resistors, and more complex cells created in a hierarchical manner such as OpAmps, TIAs, and other blocks. Pcells can be imported in Python and instantiated in larger designs. The API also supports matched placement methods, which is necessary in Analog layout. These additional methods allow for a streamlined process of combining small pcells into larger hierarchies ranging in size from differential pairs to 4-stage operational amplifiers, all fully parameterized.

These blocks are routed using several routing macros provided. The routing macros use metadata saved within the blocks called Ports. Ports represent the input or output pins on blocks. In the geometry of the layout, Ports correspond to edges of polygons and are accessed through an organized naming syntax. The names correspond to the function of the Port. For example, transistors may have three main nodes: drain, gate, and source. Each Port corresponds to an edge in layout, so Port names end with a direction indicator, North (N), East (E), South (S), or West (W). For example, if we want to refer to the west edge of the source node, we would use the port name "source_W". as shown in Fig. 3.

2.2 Strict Syntax Command Language

```
create a float parameter called width
create an int parameter called fingers
place a nmos called follower with width width and fingers=fingers
place a nmos named isrc with width width and fingers=fingers
move isrc below follower
route between follower_source_W and isrc_drain_W
```

Figure 4: Example strict syntax for a source follower.

A strict syntax file summarizes a layout topology into several text based commands. Because it can be compiled to python code, it retains all the advantages of the GLayout Python API including: PDK generic code, highly parameterizable layouts, and hierarchical blocks for easy importing. Cells can be imported either in Python format or strict syntax command format.

Each line in a strict syntax file corresponds to a set of layout operations with several possible preconfigured Python code templates. A simple example of a strict syntax file is provided in Fig. 4.

- *Create Parameter*: Parameterizing Components enhances modularity and customization, and allows for tuning layout sizing.
- *Place*: Instantiate blocks with some provided arguments, or fill in default values when left blank.
- *Move*: Reposition blocks relative to existing blocks or the origin. For example, "move m1 below m2" keeps m2 at its current position while moving m1.
- *Route*: Routing is accomplished between Ports. For instance, "route between m1_source_E and m2_source_E" will short the sources of devices m1 and m2 using the east edges as Ports.

The strict syntax is compiled to GLayout based python code (described in 2.1) using the steps as described in Fig. 5.

First a text parser is applied to the command input to identify important names, parameters, and layout blocks. The parser is implemented with a combination of regular expressions and Context

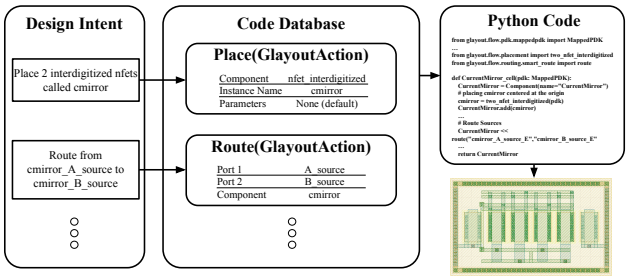


Figure 5: The command language captures design intent and uses a text parser with a database to compile Python code

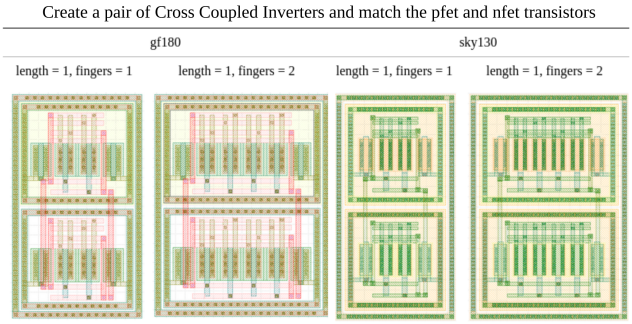


Figure 6: Cross Coupled Inverters (example is part of the training dataset).

Free Grammar (CFG) based syntaxing. CFG allow for defining a series of allowable syntaxes (called production rules). CFG style parsing was chosen for robustness in handling a variety of natural language input, and are explored with detail in [5]. This highly flexible parsing strategy allows for greater variance in valid inputs, which make strict syntax easier to learn and contributes to the high compilation rate from LLM output.

Next, the extracted information from the text parser, such as ports or parameters, are stored in a relational database which organizes information based on the command type. The syntaxer iterates through the entire command file and appends each command to the database. This step provides additional error checking, ensuring that the compiled python code will create a layout. For example, cell imports and parameters are checked to ensure the provided cells and parameters exist.

Lastly, the saved information in the database guides compilation to several preconfigured Python operations. Each command in the database is saved as a class which supports a "command.get_code" method. The get_code method chooses an appropriate Python template based on the command parameters. A wrapper Code Database

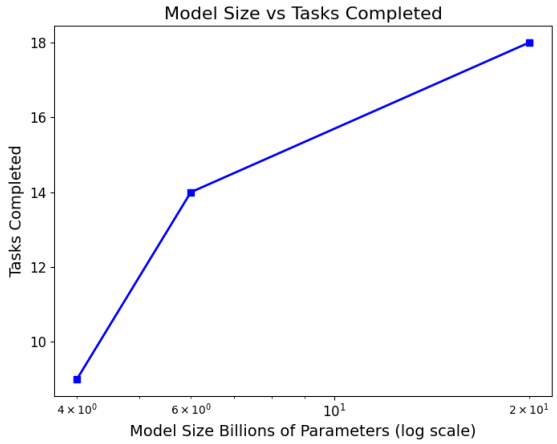


Figure 7: Larger model size directly results in better performance

class orders and combines the commands to produce a valid block generator function, including an argument for the user to provide a PDK. This block generator and all necessary imports is compiled to a Python file (which can be later imported in a larger block). The strict syntax can be compiled to any technology, provided the PDK.

2.3 Large Language Model

Using the strict syntax as an expressive text layout description, the LLM is trained to translate between a general user prompt and the strict syntax layout. The most important contributions of the LLM to this framework is the reasoning capabilities, which allows for intelligent choices in layout and highly dynamic learning capabilities. In essence, the LLM is the "mind" of GLayout, while the strict syntax is the language of choice the LLM uses to express layout design intent. There are two tasks the LLM must learn in order to produce analog layout:

(1) Understanding Layout strategies and Analog design terms. The LLM should build connections between user requests, known Analog design information, and the final layout. This task is mainly taught through in-context learning using Retrieval Augment Generation (RAG). This provides information which the LLM should learn to apply to the output layout. For example, the LLM may receive information that the transistors in a cross coupled pair should be matched. The LLM should learn how this information should influence the final layout.
(2) Describing Layout using strict syntax. This involves building an intuition for geometry especially learning relative positions of blocks and where to place components. Learning the strict syntax is complementary to this task and helps with building geometric intuition. The strict syntax also provides some common placement techniques the LLM can incorporate into its placement of blocks. This task is mainly taught through fine-tuning.

The LLM fine-tuning involves preprocessing examples to add a strict syntax reference guide and add some relevant analog design information to the prompt. The analog design information is pulled from a library of text documents using RAG. The RAG method works by constructing vector-embeddings for text documents, then computing a similarity score between the user prompt and existing text documents. The most similar text documents are returned, along with their corresponding similarity scores.

Each unique circuit can be used to create several prompts, to teach the LLM different ways of phrasing a similar request. After preprocessing, the training prompts are appended with the desired strict syntax results and the LLM is fine-tuned with loss computed on the completions only (as opposed to training on the prompt and completion). This completion only training prevents the model from over fitting to the provided context and only rewards the output strict syntax result.

3 EVALUATION

We fine-tuned and evaluated 3 LLMs: 3.8 Billion parameter Phi3 Model [10], 7 Billion parameter Mistral model [11], and 22 Billion parameter Codestral model [17]. The models all were given the same initial strict syntax context as was given during training, and

Design	Difficulty	Assessment Goal	3.8B	7B	22B
P-type Diff-Pair	easy	Different transistor type on known circuit	3	3	3
N-type Common Centroid Current mirror	easy	Different placement technique on a known circuit	1	3	3
Wilson Current Mirror (c)	easy	New placement technique, new circuit	1	2	3
Mimcap Array	easy	New placement technique	2	3	3
Class AB Stage	medium	Different placement technique on a known circuit	1	1	2
Integrator Stage	medium	Integrating known circuits, new circuit	1	1	1
Strong Arm Latch	medium	Integrating known circuits, new circuit	0	1	2
4 Stage Integrator	hard	Integrate a complex layout	0	0	1
ΔΣ Modulator	hard	Integrate a complex layout	0	0	0
			9	14	18

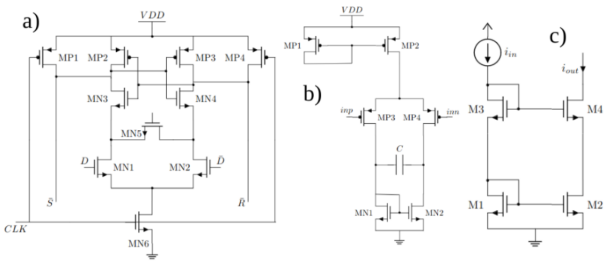


Figure 8: Evaluation pairs on which the LLM was tested. The numbers represents the number of passed tasks for that design. A design is assigned a score of 3 if the code compiles, and the layout passes DRC and LVS.

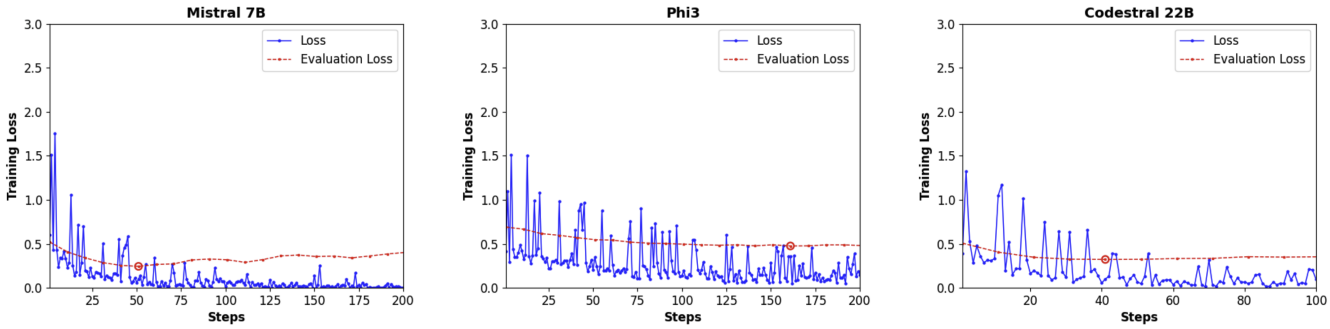


Figure 9: Fine tuning loss by step (training examples) vs Evaluation loss. After 1-2 epochs, all models overfit on the training data, resulting in less generality and performance on the evaluation data.

a previously unseen prompt was appended. These prompts either targeted a new unique layout topology for a known schematic, such as common centroid placement for a current mirror, or a completely unseen circuit, such as an integrator or strong arm latch. To evaluate the in-context learning we provided text documentation for the new circuit types. We also provided one additional prompt to guide the LLM in case of failure. The 8 evaluation examples used are categorized in Fig. 8.

Each model was trained for 4 epochs with a fixed learning rate (well past over-fitting as shown in Fig. 9), except for the 22B model which was trained for 2 epochs. All models were 8 bit quantized, and low rank adaptation (LORA) was used to reduce memory requirements while training. No other hyper-parameters or training configurations were modified for different runs; The model was the only portion changed. Fig. 7 shows the performance on the GLayout evaluation set between all 3 models vs model size. The 3.8 Billion parameter and 22 Billion Parameter models completed 10 and 18 tasks from the evaluation set respectively. We see that larger model provided a significant performance boost, but the performance gain from 7B to 22B was much less pronounced than the performance gain from 3.8B to 7B.

The models generally saw success with prompts corresponding to smaller layouts. This includes layouts such as the PMOS differential pair, shown in Table 10a. The smaller models such as 3.8B and 7B parameter models produced DRC clean layouts for layouts with a lower number of transistor placement and routing steps. These models failed to produce working strict syntax for larger layouts, where the primary point of failure was accurate placement of the components. The 22B was able to produce layouts as complex as those shown in Table 10b and Table 10c within 2 input prompts. The 22B model was also significantly better at consistently producing valid strict syntax for complex designs, which the smaller models failed to do in some cases.

4 CONCLUSION

The GLayout combined LLM approach is a scalable layout automation strategy which can achieve more complex layout with scaling both the dataset and the model size. The LLM creates DRC and LVS valid layouts on unseen 4 transistor examples, and shows performance improvements with increasing model size. The LLM achieved these layouts with less than 50 unique example circuits and trained on a single GPU for 2 hours. We showed based on results with different model sizes and known recent LLM research that this approach has potential to scale to larger layouts given a larger data set and larger model size. The model, framework, and evaluation set are completely open-source, with source code on GitHub [1], and are available for public contributions and experimentation.

ACKNOWLEDGMENTS

The authors would like to thank the open-source community for their support.

REFERENCES

[1] 2023. Glayout. https://github.com/idea-fasoc/OpenFASOC/tree/main/openfasoc/generators/glayout.

[2] 2024. GDSFactory. https://github.com/gdsfactory.

[3] Tim Ansell and Mehdi Saligane. 2020. The Missing Pieces of Open Design Enablement: A Recent History of Google Efforts : lnvited Paper. In *2020 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*. 1–8.

[4] Jacob Austin, Augustus Odena, Maxwell I. Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie J. Cai, Michael Terry, Quoc V. Le, and Charles Sutton. 2021. Program Synthesis with Large Language Models. *CoRR* abs/2108.07732 (2021). arXiv:2108.07732 https://arxiv.org/abs/2108.07732

[5] Steven Bird, Edward Loper, and Ewan Klein. 2009. Building Feature-Based Grammars. In *Natural Language Processing with Python*. O'Reilly Media Inc., Chapter 9.

[6] Yonatan Bisk, Rowan Zellers, Ronan Le Bras, Jianfeng Gao, and Yejin Choi. 2020. PIQA: Reasoning about Physical Commonsense in Natural Language. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 34. 7432–7439. https://doi.org/10.1609/aaai.v34i05.6239

[7] Jason Blocklove, Siddharth Garg, Ramesh Karri, and Hammond Pearce. 2023. Chip-Chat: Challenges and Opportunities in Conversational Hardware Design. In *2023 ACM/IEEE 5th Workshop on Machine Learning for CAD (MLCAD)*. IEEE. https://doi.org/10.1109/mlcad58807.2023.10299874

[8] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel Ziegler, Jeffrey Wu, Clemens Winter, Chris Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. 2020. Language Models are Few-Shot Learners. In *Advances in Neural Information Processing Systems*, H. Larochelle, M. Ranzato, R. Hadsell, M.F. Balcan, and H. Lin (Eds.), Vol. 33. Curran Associates, Inc., 1877–1901. https://proceedings.neurips.cc/paper_files/paper/2020/file/1457c0d6bfcb4967418bfb8ac142f64a-Paper.pdf

[9] Eric Chang, Jaeduk Han, Woorham Bae, Zhongkai Wang, Nathan Narevsky, Borivoje NikoliC, and Elad Alon. 2018. BAG2: A process-portable framework for generator-based AMS circuit design. In *2018 IEEE Custom Integrated Circuits Conference (CICC)*. 1–8. https://doi.org/10.1109/CICC.2018.8357061

[10] Marah Abdin et. al. 2024. Phi-3 Technical Report: A Highly Capable Language Model Locally on Your Phone. arXiv:2404.14219 [cs.CL]

[11] Albert Q. Jiang, Alexandre Sablayrolles, Arthur Mensch, Chris Bamford, Devendra Singh Chaplot, Diego de las Casas, Florian Bressand, Gianna Lengyel, Guillaume Lample, Lucile Saulnier, Lélio Renard Lavaud, Marie-Anne Lachaux, Pierre Stock, Teven Le Scao, Thibaut Lavril, Thomas Wang, Timothée Lacroix, and William El Sayed. 2023. Mistral 7B. arXiv:2310.06825 [cs.CL]

[12] Kishor Kunal, Meghna Madhusudan, Arvind K. Sharma, Wenbin Xu, Steven M. Burns, Ramesh Harjani, Jiang Hu, Desmond A. Kirkpatrick, and Sachin S. Sapatnekar. 2019. INVITED: ALIGN – Open-Source Analog Layout Automation from the Ground Up. In *2019 56th ACM/IEEE Design Automation Conference (DAC)*. 1–4.

[13] Yao Lai, Sungyoung Lee, Guojin Chen, Souradip Poddar, Mengkang Hu, David Z. Pan, and Ping Luo. 2024. AnalogCoder: Analog Circuit Design via Training-Free Code Generation. arXiv:2405.14918 [cs.LG]

[14] Chengjie Liu, Yijiang Liu, Yuan Du, et al. 2024. LADAC: Large Language Model-driven Auto-Designer for Analog Circuits. *TechRxiv* (January 08 2024).

[15] M. Liu, N. Pinckney, B. Khailany, and H. Ren. 2023. Verilogeval: Evaluating Large Language Models for Verilog Code Generation. In *2023 IEEE/ACM International Conference on Computer Aided Design (ICCAD)*. IEEE, 1–8.

[16] Jitesh Poojary, Ramprasath S, Sachin S. Sapatnekar, and Ramesh Harjani. 2023. Exploration of Design / Layout Tradeoffs for RF Circuits using ALIGN. In *2023 IEEE Radio Frequency Integrated Circuits Symposium (RFIC)*. 57–60. https://doi.org/10.1109/RFIC54547.2023.10186141

[17] Mistral AI Team. 2023. Codestral-22B-v0.1. https://huggingface.co/mistralai/Codestral-22B-v0.1. Accessed: 2024-06-07.

[18] Shailja Thakur, Baleegh Ahmad, Hammond Pearce, Benjamin Tan, Brendan Dolan-Gavitt, Ramesh Karri, and Siddharth Garg. 2024. VeriGen: A Large Language Model for Verilog Code Generation. *ACM Trans. Des. Autom. Electron. Syst.* 29, 3, Article 46 (apr 2024), 31 pages. https://doi.org/10.1145/3643681

[19] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed Chi, Quoc Le, and Denny Zhou. 2023. Chain-of-Thought Prompting Elicits Reasoning in Large Language Models. arXiv:2201.11903 [cs.CL]

[20] Qirui Zhang, Wenbo Duan, Tim Edwards, Tim Ansell, David Blaauw, Dennis Sylvester, and Mehdi Saligane. 2022. An Open-Source and Autonomous Temperature Sensor Generator Verified With 64 Instances in SkyWater 130 nm for Comprehensive Design Space Exploration. *IEEE Solid-State Circuits Letters* 5 (2022), 174–177. https://doi.org/10.1109/LSSC.2022.3188925

[21] Keren Zhu, Hao Chen, Mingjie Liu, and David Z. Pan. 2023. Tutorial and Perspectives on MAGICAL: A Silicon-Proven Open-Source Analog IC Layout System. *IEEE Transactions on Circuits and Systems II: Express Briefs* 70, 2 (2023), 715–720. https://doi.org/10.1109/TCSII.2022.3172869

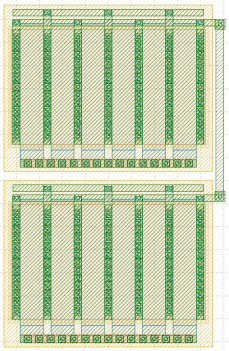
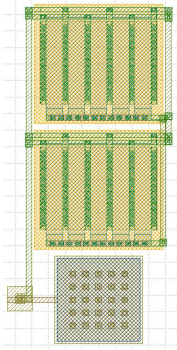
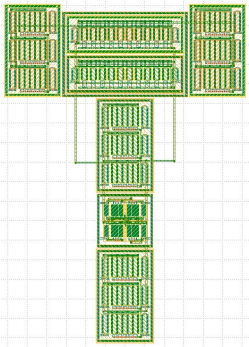
A ADDITIONAL EXAMPLES

A.1 Placement Challenges

The LLM struggles in placement for examples larger than four devices as seen in 10. This is to be expected because the training data (which is primarily composed of 2 Transistor blocks) did not include many blocks larger than 4 Transistors. As seen in 10 (a) the LLM is adept at smaller examples, while struggling with symmetric placement problems. In 10 (b) 6 transistors are overlapping because the LLM failed to place transistors on either side of the input differential pair, while pairs of transistors have been correctly placed. The strong arm latch does not suffer from this problem, and has correct placement.

A.2 Reasoning

Reasoning capabilities are demonstrated in Fig. 11 with a current mirror example. The model performs several modifications of the circuit based on user requests. The initial request (not included) is a prompt for a current mirror identical to a training prompt. As expected, the LLM produces the valid design which is similar to the training example. 11 (a) and (b) are the second prompts in two different cases following the initial current mirror prompt. In Fig. 11 (a) the user includes that it is possible to save area by removing the well tie rings (rectangular well tap surrounding the transistors) and dummy transistors, and requests the LLM to reduce the area. In prompt 11 (b) the user requests placing a second reference, resulting in 1 LVS error. This error is indicated with a red circle. Both of these tasks require the LLM to recall and combine previous information, either from training, from RAG data, or from user hints in the prompt. The LLM automatically names the produced components based on the user's request. For example, request 11 (a) was automatically named "CurrentMirrorNTypeReducedArea" and request 11 (b) was "CurrentMirrorNtypeDualRef".

LLM Prompt	Make a p-type differential pair. Parametrize everything.	Create an integrator stage. This is created using 4 pfets and two nfets with a mimcap. The nfets must have their gates shorted and one of the nfets must have their drain shorted to its gate. The sources are connected to ground. A pfet current mirror is also required …	A strong arm latch consists of a diffpair, a CrossCoupleInverters, a bridge nfet, a clkgnd nfet, and two pfets clkpwrL (west) and clkpwrR (east). place the pfets symmetrically on either side of the inverters and the other components.
Generated Strict Syntax	DiffPair create a float parameter called width create a float parameter called length create a int parameter called fingers place a pmos called diff_A with width=width, length=length, fingers=fingers, rmult=1, multipliers=1, with_substrate_tap=False, with_tie=False, with_dummy=False place a pmos called diff_B with width=width, length=length, fingers=fingers, rmult=1, multipliers=1, with_substrate_tap=False, with_tie=False, with_dummy=False move diff_B below diff_A route between diff_A_source_E and diff_B_source_E using smart_route	IntegratorStage create a float parameter called width create a float parameter called length create a int parameter called fingers ***… more parameter declarations*** place a pmos called diff_A with width=width, length=length, fingers=fingers, rmult=1, multipliers=1, with_substrate_tap=False, with_tie=False, with_dummy=False ***… more placement steps*** move nfet_B below nfet_A ***… more move steps*** route between nfet_A_drain_E and nfet_A_gate_E using smart_route ***… more route steps***	StrongArmLatch import CrossCoupledInverters create a float parameter called bridge_width create a float parameter called clkgnd_width ***… more parameter declarations*** place a diff pair called inputdiff place a CrossCoupledInverters called ccinvs with ccinvs_fingers=ccinvs_fingers, ccinvs_length=ccinvs_length ***… more placement steps*** move clkgnd below inputdiff move bridge above inputdiff ***… more move steps*** route between inputdiff_A_drain_E and bridge_drain_E using smart_route ***… more route steps***
Final Layout			

a) b) c)

Figure 10: Summary of weak-points and circuit comparison examples of the LLM.

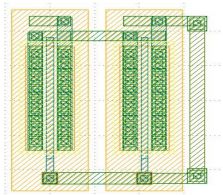
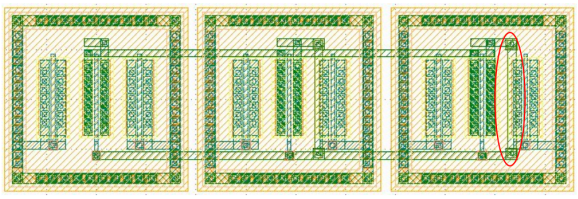
Prompt	(a) It is possible to reduce area by removing the tie rings and dummy. Reduce area of this current mirror	(b) Add another transistor to produce a second reference
Strict Syntax	CurrentMirrorNtypeReducedArea create a float parameter called width create a float parameter called length create an int parameter called fingers place a nmos called ref with width=width, length=length, fingers=fingers, rmult=1, multipliers=1, with_substrate_tap=False, with_tie=False, with_dummy=False, with_dnwell=False place a nmos called mirror with width=width, length=length, fingers=fingers, rmult=1, multipliers=1, with_substrate_tap=False, with_tie=False, with_dummy=False, with_dnwell=False move mirror to the right of ref route between ref_source_W and mirror_source_W using smart_route route between ref_gate_E and mirror_gate_E using smart_route route between mirror_drain_E and mirror_gate_E using smart_route	CurrentMirrorNtypeDualRef create a float parameter called width=3 create a float parameter called length=0.15 create an int parameter called fingers=1 place a nmos called ref1 with width=width, length=length, fingers=fingers, rmult=1, multipliers=1, with_substrate_tap=False, with_tie=True, with_dummy=True, with_dnwell=False place a nmos called ref2 with width=width, length=length, fingers=fingers, rmult=1, multipliers=1, with_substrate_tap=False, with_tie=True, with_dummy=True, with_dnwell=False place a nmos called mirror with width=width, length=length, fingers=fingers, rmult=1, multipliers=1, with_substrate_tap=False, with_tie=True, with_dummy=True, with_dnwell=False move mirror to the right of ref1 move ref2 to the right of mirror route between ref1_source_W and mirror_source_W using smart_route route between ref1_gate_E and mirror_gate_E using smart_route route between mirror_drain_E and mirror_gate_E using smart_route route between ref2_source_W and mirror_source_W using smart_route route between ref2_gate_E and mirror_gate_E using smart_route route between ref2_drain_E and mirror_gate_E using smart_route
Layout		

Figure 11: Modifying a current mirror with new requests, red circle indicates an error.