AnalogCoder: Analog Circuit Design via Training-Free Code Generation

Yao Lai[1], **Sungyoung Lee**[2], **Guojin Chen**[3], **Souradip Poddar**[2],
Mengkang Hu[1], **David Z. Pan**[2], **Ping Luo**[1]

[1] The University of Hong Kong, Hong Kong
[2] The University of Texas at Austin, Austin, Texas, United States
[3] The Chinese University of Hong Kong, Hong Kong
pluo@cs.hku.hk, dpan@ece.utexas.edu

Abstract

Analog circuit design is a significant task in modern chip technology, focusing on the selection of component types, connectivity, and parameters to ensure proper circuit functionality. Despite advances made by Large Language Models (LLMs) in digital circuit design, the complexity and scarcity of data in analog circuitry pose significant challenges. To mitigate these issues, we introduce AnalogCoder, the first training-free LLM agent for designing analog circuits through Python code generation. Firstly, AnalogCoder incorporates a feedback-enhanced flow with tailored domain-specific prompts, enabling the automated and self-correcting design of analog circuits with a high success rate. Secondly, it proposes a circuit tool library to archive successful designs as reusable modular sub-circuits, simplifying composite circuit creation. Thirdly, extensive experiments on a benchmark designed to cover a wide range of analog circuit tasks show that AnalogCoder outperforms other LLM-based methods. It has successfully designed 20 circuits, 5 more than standard GPT-4o. We believe AnalogCoder can significantly improve the labor-intensive chip design process, enabling non-experts to design analog circuits efficiently.

Code — https://github.com/laiyao1/AnalogCoder

Introduction

Analog circuits, essential for processing real-world signals such as temperature, pressure, sound, and light, are indispensable in modern integrated circuits. They facilitate accurate sensing, amplification, and filtering, which is crucial for linking digital systems with physical environments. This functionality underpins reliable data acquisition and signal processing across diverse applications.

The success of Large Language Models (LLMs) (Achiam et al. 2023) has brought new opportunities for automatic chip design (Zhong et al. 2023). Existing related research primarily focuses on two tasks: the generation and correction of Verilog codes (Blocklove et al. 2023; Chang et al. 2023; Thakur et al. 2023a,b; Fu et al. 2023; Liu et al. 2023b; Lu et al. 2024; Tsai, Liu, and Ren 2023; Liu et al. 2023c; Pei et al. 2024), and the writing of design scripts (Wu et al. 2024; Liu et al. 2023a). LLMs can convert natural language descriptions of digital circuit design tasks into Verilog code, a programming language for designing digital circuits. Once the code is generated, it can be assessed for correctness by LLMs or human experts, who attempt to fix errors by analyzing error information and simulation outputs (Blocklove et al. 2023; Thakur et al. 2023b; Tsai, Liu, and Ren 2023; Yao et al. 2024). Due to the scant representation of Verilog in the public data for training (Guo et al. 2024), LLMs may not perform as well in generating Verilog code as they do with widely-used programming languages such as C and Python, despite ongoing improvement efforts (Pei et al. 2024). Similarly, generating design flow scripts is another form of code generation, converting natural language descriptions of design requirements into script files. These scripts, written in Python or Tcl, facilitate the chip design process by invoking APIs at various stages (Ousterhout 1993; Wu et al. 2024; Liu et al. 2023a). These design flow scripts typically implement straightforward logic to transform fundamental workflows into a series of API calls. However, these works are mainly for digital circuit design, as listed in Table 1.

Analog circuit design presents significantly more challenges than digital circuit design (Johns and Martin 2008; Razavi 2000a; Allen, Dobkin, and Holberg 2011), leaving the field less explored by LLM-aided methods. The primary

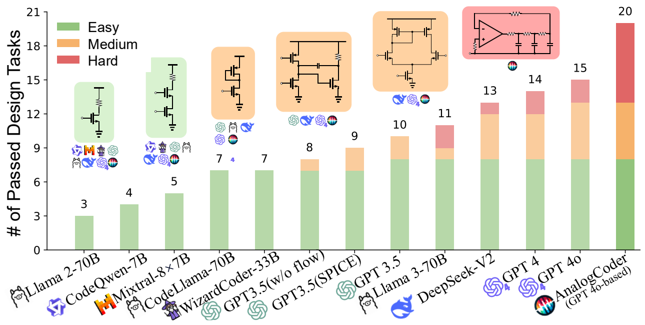


Figure 1: **Leaderboard of LLM analog circuit design.** LLMs are ranked by the number of analog circuits they design successfully. It displays several designs and lists the LLMs that successfully created them.

This work is done while the first author is a visiting student at UT Austin. The corresponding authors are Ping Luo and David Z. Pan.

Method	Auto [1]	Fix Errors [2]	Open-Source	Circuit
ChipChat	✗	✗	✓	Digital
ChipGPT	✗	✗	✗	Digital
VeriGen	✓	✗	✓	Digital
AutoChip	✓	✓	✓	Digital
VerilogEval	✓	✗	✗	Digital
RTLLM	✓	✗	✓	Digital
RTLfixer	✓	✓	✓	Digital
RTLCoder	✓	✗	✓	Digital
ChipNeMo	✓	✗	✗	Digital[3]
BetterV	✓	✗	✗	Digital
AnalogCoder	✓	✓	✓	Analog

[1] Without human involvement. [2] Automatic error fix. [3] Analog for QA questions.

Table 1: **Comparison of works.** AnalogCoder is the first LLM-based work on analog circuit design. It operates without human feedback and features automatic error correction.

challenges include: (1) *Complexity.* Unlike digital circuit design, which predominantly employs simple logic gates, analog circuits comprise diverse components such as voltage and current sources, MOSFETs, resistors, and capacitors. The complexity is further compounded by the intricate interconnections and settings required (Poddar et al. 2024). Even minor adjustments can significantly alter the circuit's functionality, potentially leading to a combinatorial explosion due to the vast search space. (2) *Abstraction level.* Digital circuit design languages like Verilog (Thomas and Moorby 2008) allow developers to write at a high level of abstraction, such as assigning functionality directly, without needing to specify the underlying hardware components like logic gates. In contrast, analog circuit design requires a direct representation of the physical components in the design code. It necessitates a more detailed and component-specific design process, making it more difficult to utilize LLM assistance effectively. For example, while a digital adder can be succinctly implemented in a single line of Verilog code, constructing an analog adder requires meticulous configuration and connection of approximately five MOSFETs and three resistors (Chaoui 1995). (3) *Corpus data volume.* Although Verilog, used for digital circuit design, constitutes a small fraction (less than 0.1%) of the repositories on GitHub, SPICE (Simulation Program with Integrated Circuit Emphasis) (Vladimirescu 1994), the predominant language for analog design, is even less common. This scarcity suggests that LLMs may find it more challenging to learn the design rules for analog circuits compared to digital ones. Thus, analog circuit design, particularly netlist generation, is a time-intensive, challenging, and error-prone process that relies heavily on the meticulous work of experienced engineers, often requiring several days to meet specific functional requirements (Dong et al. 2022b; Allen, Dobkin, and Holberg 2011). Some LLM-based circuit optimization works (Liu et al. 2024a; Yin et al. 2024) have recently been proposed, but all are based on the existing manually designed circuit netlist.

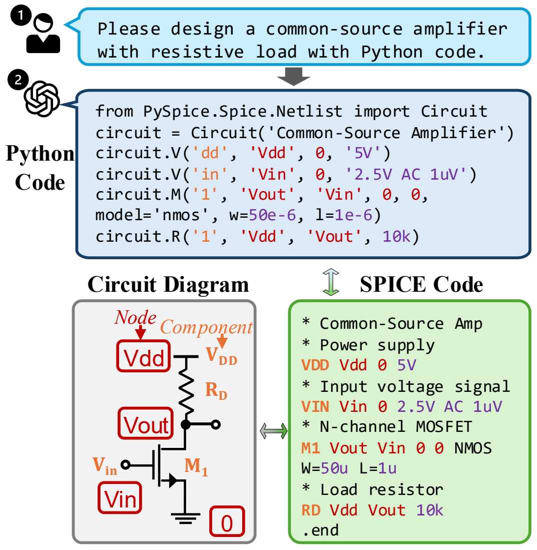
To mitigate the shortcomings of traditional manual analog circuit design and bridge the gap in LLM applications for such tasks, we introduce AnalogCoder, a novel training-free LLM-based agent that enables analog circuit design through the generation of Python code. Specifically, users can describe their desired analog circuit functionalities in natural language. AnalogCoder automatically generates the corresponding Python code for the designed circuit, leveraging the LLM's strong Python programming capabilities. To further enhance the design capabilities of LLMs, we propose domain-specific prompt engineering, feedback-enhanced design flow, and the circuit tool library, greatly increasing the success rate of design.

In this work, we prioritize the correct functionality of analog circuits, without extensive parameter optimization, already well-addressed by existing advanced methodologies (Wang et al. 2020; Lyu et al. 2018; Krylov et al. 2023). Experiments demonstrate that AnalogCoder can autonomously solve 20 out of 24 analog circuit challenges, as shown in Fig. 1, which surpasses the performance of the standard GPT-4o (15 solved) and the Llama-3 (11 solved).

This paper makes three main **contributions**: First, we introduce AnalogCoder, which, to our knowledge, is the *first* LLM-based agent for analog integrated circuit design. This agent establishes a new paradigm by generating Python code to design analog circuits. Second, we develop a feedback-enhanced design flow and a circuit tool library, significantly improving the LLM's ability to design functional analog circuits. Third, we introduce the *first* benchmark specifically designed to evaluate the ability of LLMs in designing analog circuits. This benchmark comprises 24 unique circuits, three times the number included in the ChipChat benchmark (Chang et al. 2023) and offers 40% more circuits than the VeriGen benchmark (Thakur et al. 2023a). It features detailed task descriptions, sample designs, and test-benches, enhancing resources for future research.

Preliminary

Analog Circuits. Unlike digital circuits, which exclusively process discrete binary signals, analog circuits manage continuous-valued signals, enabling a diverse array of functionalities (Razavi 2000b). For example, an analog amplifier, as depicted in Fig. 2, is engineered to enhance the amplitude of an input signal, expressed as $V_{out}(t) = A_v \times V_{in}(t)$, where $V_{in}(t)$ and $V_{out}(t)$ denote the time-variant behavior of the input and output voltage signals, respectively, and A_v represents the voltage gain of the amplifier. Moreover, operational amplifiers (op-amps) are high-gain voltage amplifiers with differential inputs, featuring a non-inverting input V_{inp} and an inverting input V_{inn}. The output of the op-amp is expressed as $V_{out}(t) = A_v \times [V_{inp}(t) - V_{inn}(t)]$, enabling it to be configured to perform a variety of analog signal operations, including integration, differentiation, addition, and subtraction. When configured as an adder, for instance, the operational amplifier can implement the function $V_{out}(t) = -[V_{in1}(t)+V_{in2}(t)]$. Moreover, when set up as an integrator, it can integrate the input voltage signal, yielding an output given by $V_{out}(t) = -\int V_{in}(t)\,dt/\tau$, where τ represents the time constant associated with the resistance and capacitance values within the circuit. These analog circuits demonstrate how analog operations transform input signals



```
Please design a common-source amplifier
with resistive load with Python code.
```

```
from PySpice.Spice.Netlist import Circuit
circuit = Circuit('Common-Source Amplifier')
circuit.V('dd', 'Vdd', 0, '5V')
circuit.V('in', 'Vin', 0, '2.5V AC 1uV')
circuit.M('1', 'Vout', 'Vin', 0, 0,
model='nmos', w=50e-6, l=1e-6)
circuit.R('1', 'Vdd', 'Vout', 10k)
```

```
* Common-Source Amp
* Power supply
VDD Vdd 0 5V
* Input voltage signal
VIN Vin 0 2.5V AC 1uV
* N-channel MOSFET
M1 Vout Vin 0 0 NMOS
W=50u L=1u
* Load resistor
RD Vdd Vout 10k
.end
```

Figure 2: **Circuit Representation.** We input the task into the LLM, and it outputs Python code. Three interconvertible representations of the designed circuit: (1) Circuit diagrams typically take days for human experts to design due to the intricate and demanding process of selecting and connecting components. (2) SPICE code with formatted netlists. (3) Python code with the PySpice library, achieving circuits equivalent to those generated by SPICE code. Experts manually sketch circuit diagrams due to the complexity and non-intuitive nature of coding.

into output signals, accomplishing computations more efficiently than clock-dependent digital circuits. Testing an analog circuit involves applying a specific input and verifying that the output aligns with expected standards to ensure its correct operation. After simulation, attributes such as gain are identified as specifications. These specifications are essential for validating the circuit's performance against its design requirements.

Code Representation for Circuits. To facilitate the description and simulation of analog circuit designs, SPICE (Vladimirescu 1994) has been introduced. Often used as a programming language, this tool allows designers to specify the complex interconnections between electronic components within a circuit. With SPICE codes, the behavior of circuits can be accurately simulated and analyzed, with each component, such as resistors, capacitors, and voltage or current sources, carefully itemized and connected in a notation recognized industry-wide. In the SPICE syntax, the fundamental constructs are elements and nodes (see Fig. 2). The elements refer to various electronic components like resistors and transistors, while nodes denote the points at which these elements are interconnected. As shown in the circuit diagram in Fig. 2, the amplifier comprises four elements: two voltage sources, V_{dd} and V_{in}, for the power supply and signal input, respectively; one N-channel MOSFET, M_1, for signal amplification; and one resistive load, R_D. Four lines in the SPICE code describe these elements. Each line in the SPICE code starts with the element name, followed by the names of the nodes to which the element is connected. For instance, the resistor R_D is connected between nodes V_{dd} and V_{out}. The corresponding SPICE code line is '`RD Vdd Vout 10k`', where '`10k`' denotes $10\,\mathrm{k}\Omega$. Specifically, since a MOSFET has four connection nodes, the corresponding code line will include four node labels delineating the drain, gate, source, and bulk connections. PySpice (Salvaire 2021) integrates SPICE code with the Python programming language, leveraging Python's user-friendly syntax and robust ecosystem to simplify circuit simulation and data processing, as demonstrated in the Python code in Fig. 2. This integration allows for more accessible and efficient design workflows, broadening the usability of SPICE. Since LLMs excel at Python programming (Khan et al. 2023; Zheng et al. 2023), we chose Python with the PySpice library to automate the creation of circuits, replacing the tedious manual process.

Our Approach

Method Overview. AnalogCoder is an LLM-based agent that interprets task descriptions in natural language to generate Python code, automatically representing functionally correct analog circuits. To enhance the agent's design capabilities, we implemented a comprehensive methodology as shown in Fig. 3, including prompt engineering, a feedback-enhanced design flow, and a circuit tool library. Prompt engineering enhances the agent's design thinking through strategic, problem-solving prompts. The feedback-enhanced design flow uses multiple checks to provide error feedback to the agent, facilitating the correction of failed designs by LLMs. The circuit tool library, a modular sub-circuit repository, systematically organizes designed circuits as tools, enabling straightforward retrieval and reuse by LLMs for complex circuit designs.

Prompt Engineering. We initially established a well-crafted design prompt to maximize the design capabilities of LLMs. Our approach to prompt engineering encompasses three main aspects: (1) *programming language selection*, (2) *in-context learning* (Dong et al. 2022a), and (3) *Chain-of-Thought* (Wei et al. 2022). Despite the capability of LLMs to generate code in multiple programming languages, their performance in Python surpasses that in most others (Cassano et al. 2023; Zheng et al. 2023). Additionally, many prominent code-generating LLMs, such as CodeLlama (Roziere et al. 2023) and WizardCoder (Luo et al. 2023), are primarily fine-tuned on Python datasets, indicating a bias towards Python. Conversely, the training datasets for most LLMs, which are based on GitHub, do not contain sufficient data on SPICE code (Guo et al. 2024). Therefore, to mitigate this limitation, we directly prompt the LLM to generate executable Python code compatible with the PySpice library. Furthermore, we integrate in-context learning (Dong et al. 2022a) to enhance circuit design, providing a detailed example of a two-stage amplifier with active and resistor loads as one-shot learning (Brown et al. 2020). This example facilitates the LLM's learning and imitation and standardizes its output, minimizing errors. All design tasks are distinct from

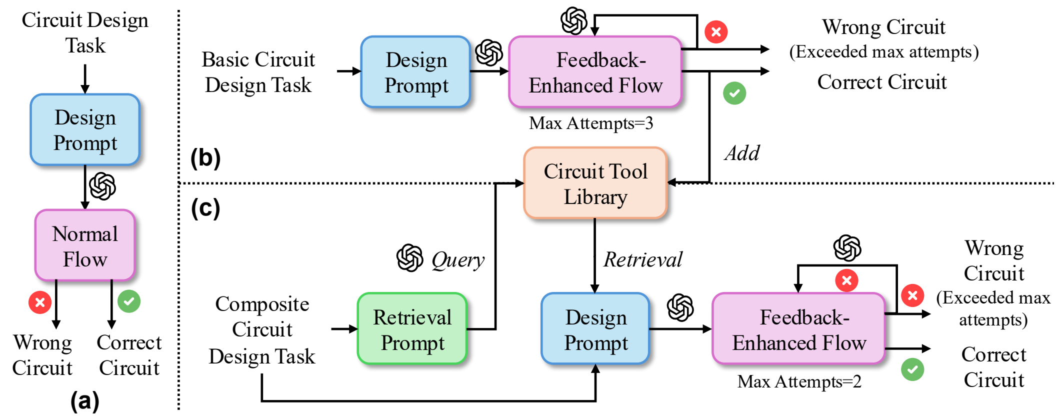
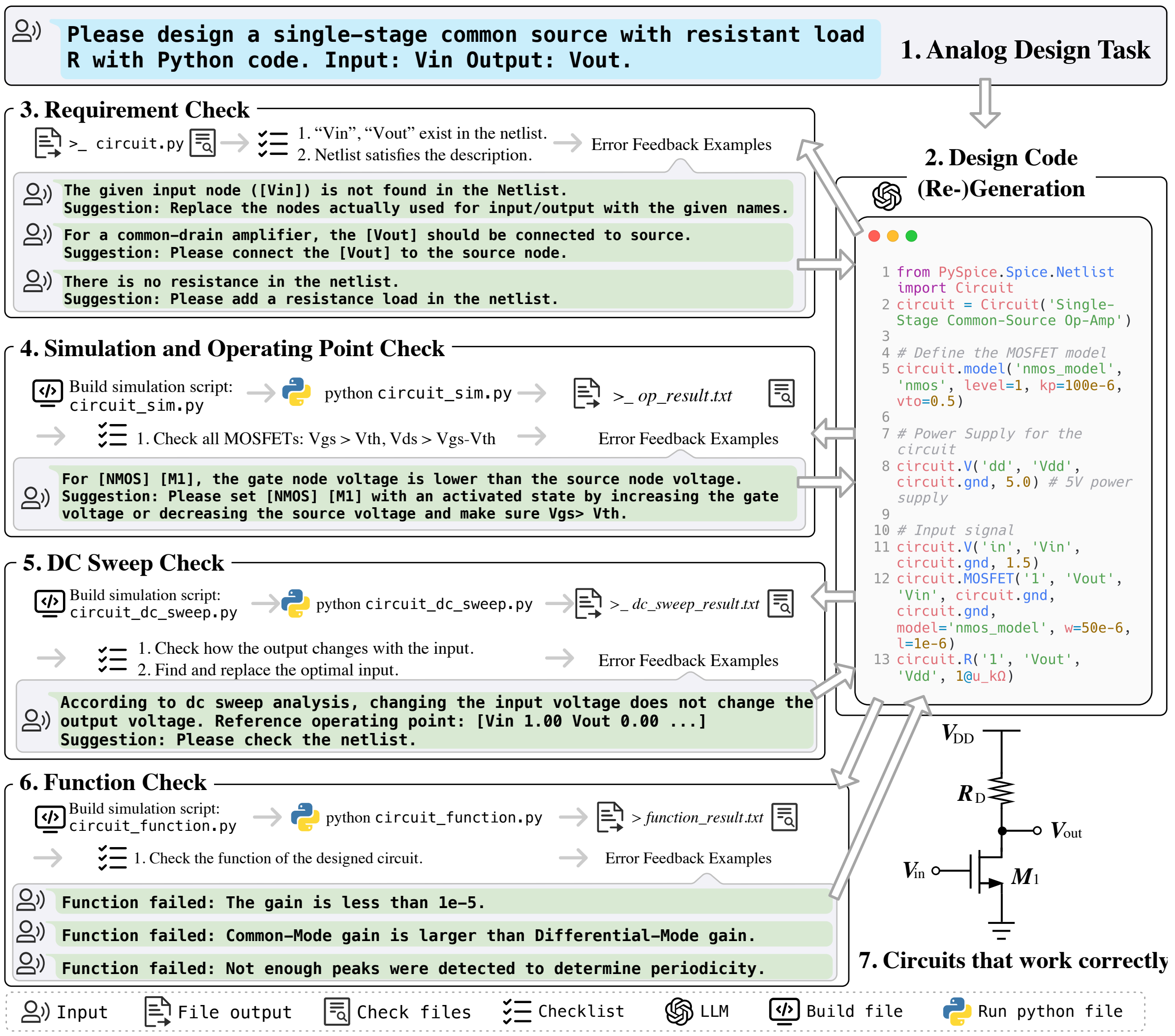


Figure 3: **Method Overview. (a) Standard method. (b) Our method for basic circuit.** Input the design prompts to the feedback-enhanced design flow, enabling the automated error fix with LLMs. Successfully designed circuits are added to the circuit tool library, while failed designs are returned to the LLM for automatic fixing. **(c) Our method for composite circuit.** The process adds a step of querying the library to retrieve invocation methods for subcircuits, which are then integrated into the design prompt to facilitate the design of composite circuits.

the provided example to maintain evaluation fairness. Additionally, the Chain-of-Thought strategy (Wei et al. 2022) involves prompting the LLM to generate a detailed design plan, including necessary components and their interconnections. This plan subsequently guides the generation of the corresponding design code, simplifying the design task significantly. Complete prompts can be seen in the Appendix.

Feedback-Enhanced Design Flow. Various errors are often observed in the code generated by LLMs. Consequently, guiding LLMs to correct the generated codes based on error messages is crucial. Numerous studies (Hong et al. 2023; Chen et al. 2023; Olausson et al. 2023; Tsai, Liu, and Ren 2023; Thakur et al. 2023a) have suggested that providing LLMs with relevant error information helps LLMs fix faulty code. However, for the analog circuit design, besides the runtime errors that may occur when executing SPICE simulations, additional verification of circuit-related information is necessary to ensure the correctness of the design. In analog circuit design, when a design fails, we return either runtime errors from the Python code or circuit-specific test errors to the LLM, as illustrated in Fig. 4. We divide the feedback-enhanced flow into four stages: (1) *requirement check*, (2) *simulation and operating point check*, (3) *DC sweep check*, and (4) *function check*. The *requirement check* is to verify whether the generated code meets the basic design requirements, such as the presence of requisite inputs and outputs, and the inclusion of essential circuit components. The *simulation and operating point check* initially assesses whether the generated analog circuit can successfully execute simulations, aiming to identify issues such as floating nodes and other potential errors. Once the simulation passes, the static operating point voltages of nodes are achieved. Examining these operating point voltages ensures that the MOSFET transistors are in their correct operational states. The *DC sweep check* performs a direct current (DC) analysis by changing the voltage at the input nodes and observing the corresponding changes at the output nodes to verify the integrity of the signal path from input to output. This method also helps identify the optimal bias voltage, increasing the success rate of the design. The *function check* simulates specific input waveforms and observes the outputs to verify the analog circuit's fundamental functionalities. The simulation may involve DC, AC (alternating current), or transient analyses depending on the circuit types. For any errors occurring in these checks, the relevant error information is returned to the LLM, which then prompts it to *regenerate complete and correct codes*. Due to limitations in the LLMs' code repair capabilities, we allow up to three times code generations.

Circuit Tool Library. As analog circuit design tasks become more complex and the implementation code grows more intricate, it becomes increasingly challenging for LLMs to generate correct circuits. To address this complexity, basic circuits can be encapsulated into subcircuit modules in the SPICE code, facilitating their integration into more composite assemblies. Building on this modular approach and inspired by the tool-based LLM studies (Wang et al. 2023; Qin et al. 2023), we adopted a circuit tool library that stores correctly designed subcircuits for easy reuse in more complex designs. As illustrated in Fig. 5, our approach involves two main processes: adding circuits to the library (top) and retrieving circuits from the library (bottom). After an LLM-based agent completes a basic circuit design task, we add the circuit codes and the specifications from the simulation results to the circuit tool library. If a circuit task has been successfully completed multiple times, store the optimal circuit design based on the key specification, such as gain. The task descriptions and circuit information are stored as keys for queries, while the codes and calling methods are stored as values. In composite circuit design, the task description is used to formulate a query prompt, enabling the retrieval of the requisite subcircuit tools by LLMs.


1. Analog Design Task
2. Design Code (Re-)Generation
3. Requirement Check
>_ circuit.py
1. "Vin", "Vout" exist in the netlist.
2. Netlist satisfies the description.
Error Feedback Examples
4. Simulation and Operating Point Check
Build simulation script: circuit_sim.py
python circuit_sim.py
>_ op_result.txt
1. Check all MOSFETs: Vgs > Vth, Vds > Vgs-Vth
Error Feedback Examples
5. DC Sweep Check
Build simulation script: circuit_dc_sweep.py
python circuit_dc_sweep.py
>_ dc_sweep_result.txt
1. Check how the output changes with the input.
2. Find and replace the optimal input.
Error Feedback Examples
6. Function Check
Build simulation script: circuit_function.py
python circuit_function.py
> function_result.txt
1. Check the function of the designed circuit.
Error Feedback Examples
V_{DD}
R_D
V_{out}
V_{in}
M_1
7. Circuits that work correctly
Input
File output
Check files
Checklist
LLM
Build file
Run python file

```
Please design a single-stage common source with resistant load
R with Python code. Input: Vin Output: Vout.
```

```
The given input node ([Vin]) is not found in the Netlist.
Suggestion: Replace the nodes actually used for input/output with the given names.
```

```
For a common-drain amplifier, the [Vout] should be connected to source.
Suggestion: Please connect the [Vout] to the source node.
```

```
There is no resistance in the netlist.
Suggestion: Please add a resistance load in the netlist.
```

```
For [NMOS] [M1], the gate node voltage is lower than the source node voltage.
Suggestion: Please set [NMOS] [M1] with an activated state by increasing the gate
voltage or decreasing the source voltage and make sure Vgs> Vth.
```

```
According to dc sweep analysis, changing the input voltage does not change the
output voltage. Reference operating point: [Vin 1.00 Vout 0.00 ...]
Suggestion: Please check the netlist.
```

```
Function failed: The gain is less than 1e-5.
```

```
Function failed: Common-Mode gain is larger than Differential-Mode gain.
```

```
Function failed: Not enough peaks were detected to determine periodicity.
```

```python
from PySpice.Spice.Netlist import Circuit
circuit = Circuit('Single-Stage Common-Source Op-Amp')

# Define the MOSFET model
circuit.model('nmos_model', 'nmos', level=1, kp=100e-6, vto=0.5)

# Power Supply for the circuit
circuit.V('dd', 'Vdd', circuit.gnd, 5.0) # 5V power supply

# Input signal
circuit.V('in', 'Vin', circuit.gnd, 1.5)
circuit.MOSFET('1', 'Vout', 'Vin', circuit.gnd, circuit.gnd, model='nmos_model', w=50e-6, l=1e-6)
circuit.R('1', 'Vout', 'Vdd', 1@u_kΩ)
```

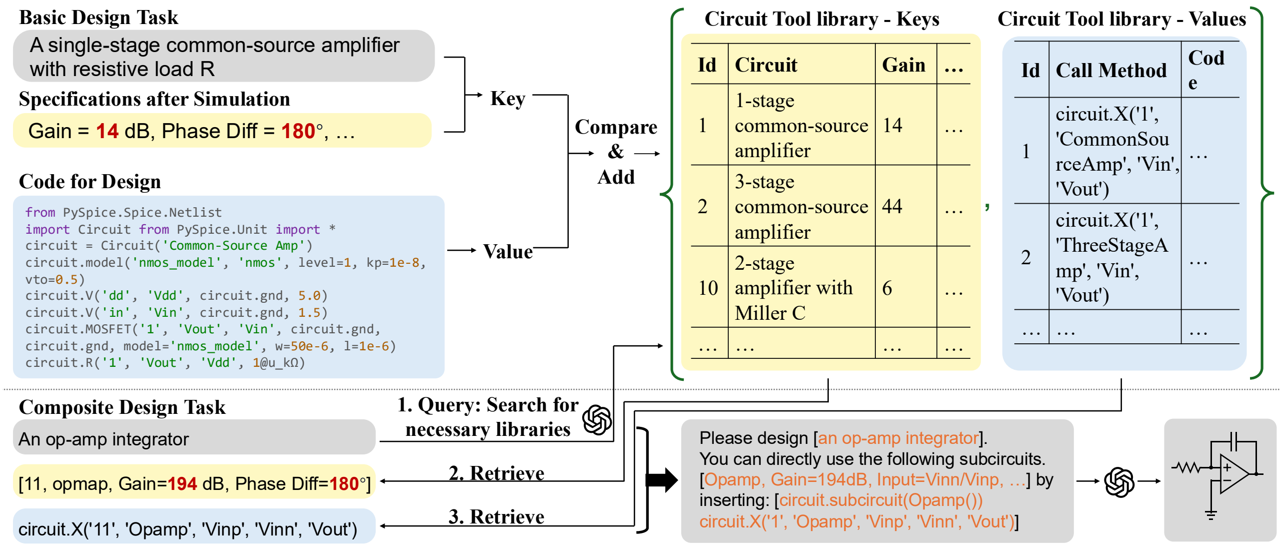
Figure 4: **Feedback-Enhanced Design Flow.** The flow facilitates autonomous error correction in designs by the LLM agent without human intervention. Error messages are returned to the LLM to assist in refining the design. Suggestions are saved as templates according to error types. The entire flow is adaptable to nearly all categories of analog circuits.

The agent initially retrieves the indices of the required subcircuits and then uses these indices to fetch all corresponding specifications and calling methods. This information is then integrated with the task description and automatically re-entered into the LLM to design the circuit. At this stage, the agent uses the retrieved subcircuits' calling methods to directly integrate them into the code, thereby designing composite circuits. As shown in Fig. 5, when designing an op-amp integrator, the LLM queries and retrieves the index corresponding to the required subcircuit, a single-stage op-amp. Subsequently, the task description, along with the pertinent information of this subcircuit, is input into the LLM, which then generates the design code for the op-amp integrator.

Fine-tuning. Due to the scarcity of datasets for analog circuits and inspired by GPT-assisted data generation (Liu et al. 2024b), we collected samples of successful circuit designs created by GPT-3.5, GPT-4o, and Llama-3 to fine-tune GPT-3.5 by the provided API. We gathered successful designs for each task and clustered them into three categories using text vectorization (Sparck Jones 1972). One design from each category was selected and paired with the input prompt to form initial pairs, then refined through text filtering to create the fine-tuning data.

Experiments

We extensively evaluate the capability of LLMs in analog circuit design, including Mixtral-7×8B (Jiang et al. 2024), CodeLlama-70B-Instruct (Roziere et al. 2023), Wizardcoder-33B-V1.1 (Luo et al. 2023), Llama3-70B (AI@Meta 2024), DeepSeek-V2 (DeepSeek-AI 2024), GPT-3.5-turbo (Brown et al. 2020), GPT-4-turbo (Achiam et al. 2023) and GPT-4o. CodeLlama and WizardCoder are code generation LLMs, fine-tuned on Llama2 (Touvron et al. 2023) and StarCoder (Li et al. 2023), respectively. Llama-


Basic Design Task
A single-stage common-source amplifier with resistive load R
Specifications after Simulation
Gain = 14 dB, Phase Diff = 180°, …
Code for Design
Key
Value
Compare & Add
Circuit Tool library - Keys
Circuit Tool library - Values
Composite Design Task
An op-amp integrator
1. Query: Search for necessary libraries
[11, opmap, Gain=194 dB, Phase Diff=180°]
2. Retrieve
circuit.X('11', 'Opamp', 'Vinp', 'Vinn', 'Vout')
3. Retrieve

```
from PySpice.Spice.Netlist
import Circuit from PySpice.Unit import *
circuit = Circuit('Common-Source Amp')
circuit.model('nmos_model', 'nmos', level=1, kp=1e-8,
vto=0.5)
circuit.V('dd', 'Vdd', circuit.gnd, 5.0)
circuit.V('in', 'Vin', circuit.gnd, 1.5)
circuit.MOSFET('1', 'Vout', 'Vin', circuit.gnd,
circuit.gnd, model='nmos_model', w=50e-6, l=1e-6)
circuit.R('1', 'Vout', 'Vdd', 1@u_kΩ)
```

Id	Circuit	Gain	…
1	1-stage common-source amplifier	14	…
2	3-stage common-source amplifier	44	…
10	2-stage amplifier with Miller C	6	…
…	…	…	…

Id	Call Method	Code
1	circuit.X('1', 'CommonSourceAmp', 'Vin', 'Vout')	…
2	circuit.X('1', 'ThreeStageAmp', 'Vin', 'Vout')	…
…	…	…

Please design [an op-amp integrator].
You can directly use the following subcircuits.
[Opamp, Gain=194dB, Input=Vinn/Vinp, …] by inserting: [circuit.subcircuit(Opamp())
circuit.X('1', 'Opamp', 'Vinp', 'Vinn', 'Vout')]

Figure 5: **Circuit Tool library. Top:** Addition of new tools derived from successfully designed basic circuits. Here, descriptions and specifications are keys, while design codes are stored as values. **Bottom:** Retrieval of tools from the library for designing composite circuits. The process begins with the LLM querying necessary tools using the task description. Subsequently, the keys and values of the retrieved tools, with the task description, are employed as prompts for circuit design.

3 and DeepSeek-V2 are the newest open-source general LLMs. WizardCoder, DeepSeek-V2, and Llama-3 are LLMs that outperformed GPT-3.5 on the HumanEval (Chen et al. 2021) coding tasks (Liu et al. 2024c). Additional models can be seen in the Appendix. Open-source models were evaluated on 4 Nvidia A100 GPUs.

Metrics. We use 'Pass@k' (Kulal et al. 2019) (k=1, 5), a metric widely used in code generation tasks (Roziere et al. 2023; Li et al. 2022; Luo et al. 2023; Liu et al. 2024c; Guo et al. 2024), as the main evaluation metric. It is defined as the ratio of correct generations within k independent trials, with higher values being better. We conduct n trials $(n \geq k)$, and compute $Pass@k = 1 - \binom{n-c}{k}/\binom{n}{k}$, where c is the number of successful trials. For open-source LLMs and GPT-3.5, we set $n = 30$; for fine-tuned GPT-3.5, GPT-4, and GPT-4o, $n = 15$. '# Solved' refers to the count of distinct tasks for which a circuit design is successfully achieved at least once in n trials.

Benchmark. We have developed a comprehensive benchmark of analog circuit design tasks, detailed in Table 2. The difficulty of these tasks is determined by the number of components and the complexity of their connections. Tasks 1-15 are basic circuits, while 16-24 are composite circuits.

Main Results. Table 3 compares our LLM agent, AnalogCoder, based on GPT-4o and incorporates prompt engineering, flow feedback, and a circuit tool library with other LLM-based methods. To ensure a fair comparison and highlight the tool library's impact, we applied our strategies across all LLMs but specifically excluded the circuit tool library from GPT-4o to isolate its effects. The results indicate that Llama-3 and DeepSeek-V2, the latest open-source models, demonstrate a marginally superior capability in circuit design compared to GPT-3.5. However, other open-source models still exhibit a certain gap compared to GPT-3.5, although some surpassed GPT-3.5 in normal Python coding tasks (Liu et al. 2024c). This is primarily because circuit design requires coding skills and specific background knowledge; hence, general LLMs tend to perform better. GPT-4o is still the best LLM for analog circuit design, generally consistent with other findings on its performance in coding tasks (Luo et al. 2023; Bai et al. 2023; Liu et al. 2024c; Guo et al. 2024). Benefiting from the library, GPT-4o and Llama-3 can further utilize existing circuits to design more challenging composite circuits, enhancing their design capabilities.

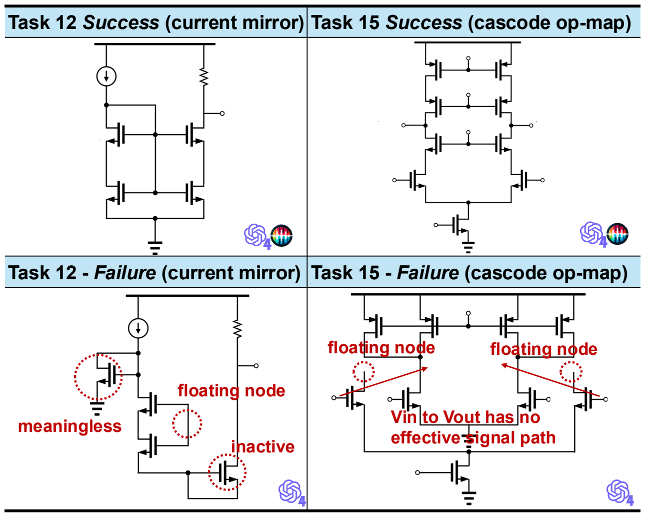

Task 12 Success (current mirror)
Task 15 Success (cascode op-map)
Task 12 - Failure (current mirror)
Task 15 - Failure (cascode op-map)
meaningless
floating node
inactive
floating node
floating node
Vin to Vout has no effective signal path

Figure 6: **Visualization for successful and failed designs.**

Id	**Type**	**Circuit Description**	**Id**	**Type**	**Circuit Description**
1	Amplifier	Common-source amp. with R load	13	Opamp	Common-source op-amp with R loads
2	Amplifier	3-stage common-source amplifier with R loads	14	Opamp	2-stage op-amp with active loads
3	Amplifier	Common-drain amp. with R load	15	Opamp	Cascode op-amp with cascode loads
4	Amplifier	Common-gate amp. with R load	16	Oscillator	RC Shift oscillator
5	Amplifier	Cascode amp. with R load	17	Oscillator	Wien Bridge oscillator
6	Inverter	NMOS inverter with R load	18	Integrator	Op-amp integrator
7	Inverter	Logical inverter with NMOS and PMOS	19	Differentiator	Op-amp differentiator
8	Current Mirror	NMOS constant current source with R load	20	Adder	Op-amp adder
9	Amplifier	Common-source amp. with diode-connected load	21	Subtractor	Op-amp subtractor
10	Amplifier	2-stage amplifier with Miller compensation C	22	Schmitt trigger	Non-inverting Schmitt trigger
11	Opamp	Op-amp with active current mirror loads	23	VCO	Voltage-Controlled Oscillator
12	Current Mirror	Cascode current mirror	24	PLL	Phase-Locked Loop

Table 2: **Benchmark Descriptions.** Different difficulties are distinguished by background colors (easy, medium, and hard).

Model	**CodeLlama-70B**		**WizardCoder-33B**		**DeepSeek-V2**		**Llama3-70B**		**GPT3.5**		**GPT4o (w/o tool)**		**AnalogCoder**	
Task ID	Pass@1	Pass@5	Pass@1	Pass@5	Pass@1	Pass@5	Pass@1	Pass@5	Pass@1	Pass@5	Pass@1	Pass@5	Pass@1	Pass@5
1	20.0	70.2	93.3	**100.0**	**100.0**	**100.0**	93.3	**100.0**	86.7	**100.0**	**100.0**	**100.0**	**100.0**	**100.0**
2	3.3	16.7	13.3	53.8	93.3	**100.0**	20.0	70.2	70.0	99.9	**100.0**	**100.0**	**100.0**	**100.0**
3	0.0	0.0	0.0	0.0	83.3	**100.0**	90.0	**100.0**	3.3	16.7	**100.0**	**100.0**	**100.0**	**100.0**
4	3.3	16.7	10.0	43.3	70.0	99.9	83.3	**100.0**	50.0	97.9	**100.0**	**100.0**	**100.0**	**100.0**
5	3.3	16.7	13.3	53.8	76.7	**100.0**	20.0	70.2	10.0	43.3	**100.0**	**100.0**	**100.0**	**100.0**
6	23.3	76.4	13.3	53.8	**100.0**	**100.0**	**100.0**	**100.0**	73.3	**100.0**	**100.0**	**100.0**	**100.0**	**100.0**
7	10.0	43.3	6.7	31.0	**100.0**	**100.0**	**100.0**	**100.0**	76.7	**100.0**	**100.0**	**100.0**	**100.0**	**100.0**
8	13.3	53.8	20.0	70.2	96.7	**100.0**	93.3	**100.0**	66.7	99.8	**100.0**	**100.0**	**100.0**	**100.0**
9	0.0	0.0	0.0	0.0	93.3	**100.0**	0.0	0.0	30.0	85.7	**100.0**	**100.0**	**100.0**	**100.0**
10	0.0	0.0	0.0	0.0	**100.0**	**100.0**	83.3	**100.0**	46.7	96.9	**100.0**	**100.0**	**100.0**	**100.0**
11	0.0	0.0	0.0	0.0	3.3	16.7	0.0	0.0	0.0	0.0	**100.0**	**100.0**	**100.0**	**100.0**
12	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	**13.3**	**57.1**	**13.3**	**57.1**
13	0.0	0.0	0.0	0.0	3.3	16.7	0.0	0.0	0.0	0.0	**100.0**	**100.0**	**100.0**	**100.0**
14	0.0	0.0	0.0	0.0	6.7	31.0	0.0	0.0	0.0	0.0	**73.3**	**100.0**	**73.3**	**100.0**
15	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	**13.3**	**57.1**	**13.3**	**57.1**
16	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	**6.7**	**33.3**
18	0.0	0.0	0.0	0.0	0.0	0.0	3.3	16.7	0.0	0.0	0.0	0.0	**100.0**	**100.0**
19	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	**60.0**	**99.8**
20	0.0	0.0	0.0	0.0	0.0	0.0	3.3	16.7	0.0	0.0	0.0	0.0	**100.0**	**100.0**
21	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	**20.0**	**73.6**
17, 22-24	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
Avg	3.2	12.2	7.1	16.9	38.6	44.3	28.8	36.4	21.4	35.0	54.2	58.9	**66.1**	**75.9**
# Solved	7	7	7	7	13	13	11	11	10	10	15	15	**20**	**20**

Table 3: **Main results.** LLMs have been enhanced by prompt engineering, design flow feedback, and the tool library.

Ablations. We evaluated the effectiveness of various components within our approach using the GPT-3.5 model, with results presented in Table 4. Specifically, "GPT-3.5 (w/ SPICE)" refers to GPT-3.5 in which the LLM is prompted to generate SPICE codes rather than Python. The variants "GPT-3.5 (w/o ICL)" and "GPT-3.5 (w/o CoT)" explore the impact on performance when omitting in-context learning and Chain-of-Thought reasoning from the prompts, respectively. Furthermore, "GPT-3.5 (w/o flow)" indicates a setup in which our proposed design flow was not utilized, and only the first generated codes were applied for functional testing. The findings consistently show that removing these components leads to a decrease in design performance.

Method	**Pass@1**	**Pass@5**	**# Solved**
GPT-3.5 w/ SPICE	13.9	26.9	9
GPT-3.5 w/o ICL	8.1	18.5	7
GPT-3.5 w/o CoT	19.4	26.3	8
GPT-3.5 w/o flow	12.8	25.3	8
GPT-3.5	21.4	35.0	10
GPT-3.5 Finetune	28.1	39.6	10

Table 4: **Ablations and fine-tuning.**

Fine-tuning. We employed a 3-fold cross-validation for fine-tuning evaluation, using two subsets of design tasks for fine-tuning and the remaining one for testing. Fine-tuning was performed using the GPT-3.5 API with two epochs. The results are shown in Table 4. Fine-tuned GPT-3.5 generally performs better on design tasks, as fine-tuning helps standardize design outputs through correct examples and reduces common syntax and design errors. However, due to the inherent limitations of the GPT-3.5 base model, fine-tuned models struggle to design additional circuits when data are limited. Further details are provided in the appendix.

Visualization. Several circuit design diagrams are in Fig. 6, with bottom icons indicating the source LLMs.

Conclusion

This paper introduces AnalogCoder, a training-free LLM agent for automatic analog circuit design. By innovatively transforming the task into Python code generation, it significantly reduces the complexity faced by LLMs. It is equipped with crafted prompts, a feedback-enhanced design flow, and a circuit tool library, effectively enhancing the preformance.

Acknowledgments

This paper is partially supported by the National Key R&D Program of China No.2022ZD0161000, the General Research Fund of Hong Kong No.17200622 and 17209324, Samsung, SRC under task 3160.007, and an equipment donation from Nvidia to UT Austin.

References

Achiam, J.; Adler, S.; Agarwal, S.; Ahmad, L.; Akkaya, I.; Aleman, F. L.; Almeida, D.; Altenschmidt, J.; Altman, S.; Anadkat, S.; et al. 2023. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774*.

AI@Meta. 2024. Llama 3 Model Card.

Allen, P. E.; Dobkin, R.; and Holberg, D. R. 2011. *CMOS analog circuit design*. Elsevier.

Bai, J.; Bai, S.; Chu, Y.; Cui, Z.; Dang, K.; Deng, X.; Fan, Y.; Ge, W.; Han, Y.; Huang, F.; et al. 2023. Qwen technical report. *arXiv preprint arXiv:2309.16609*.

Blocklove, J.; Garg, S.; Karri, R.; and Pearce, H. 2023. Chip-chat: Challenges and opportunities in conversational hardware design. In *ACM/IEEE 5th Workshop on Machine Learning for CAD (MLCAD)*, 1–6. IEEE.

Brown, T.; Mann, B.; Ryder, N.; Subbiah, M.; Kaplan, J. D.; Dhariwal, P.; Neelakantan, A.; Shyam, P.; Sastry, G.; Askell, A.; et al. 2020. Language models are few-shot learners. *Advances in neural information processing systems (NeurIPS)*, 33: 1877–1901.

Cassano, F.; Gouwar, J.; Nguyen, D.; Nguyen, S.; Phipps-Costin, L.; Pinckney, D.; Yee, M.-H.; Zi, Y.; Anderson, C. J.; Feldman, M. Q.; et al. 2023. MultiPL-E: a scalable and polyglot approach to benchmarking neural code generation. *IEEE Transactions on Software Engineering (TSE)*.

Chang, K.; Wang, Y.; Ren, H.; Wang, M.; Liang, S.; Han, Y.; Li, H.; and Li, X. 2023. Chipgpt: How far are we from natural language hardware design. *arXiv preprint arXiv:2305.14019*.

Chaoui, H. 1995. CMOS analogue adder. *Electronics Letters*, 31(3): 180–181.

Chen, M.; Tworek, J.; Jun, H.; Yuan, Q.; Pinto, H. P. d. O.; Kaplan, J.; Edwards, H.; Burda, Y.; Joseph, N.; Brockman, G.; et al. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*.

Chen, X.; Lin, M.; Schärli, N.; and Zhou, D. 2023. Teaching large language models to self-debug. *arXiv preprint arXiv:2304.05128*.

DeepSeek-AI. 2024. DeepSeek-V2: A Strong, Economical, and Efficient Mixture-of-Experts Language Model. arXiv:2405.04434.

Dong, Q.; Li, L.; Dai, D.; Zheng, C.; Wu, Z.; Chang, B.; Sun, X.; Xu, J.; and Sui, Z. 2022a. A survey on in-context learning. *arXiv preprint arXiv:2301.00234*.

Dong, Z.; Cao, W.; Zhang, M.; Tao, D.; Chen, Y.; and Zhang, X. 2022b. CktGNN: Circuit Graph Neural Network for Electronic Design Automation. In *The Eleventh International Conference on Learning Representations (ICLR)*.

Fu, Y.; Zhang, Y.; Yu, Z.; Li, S.; Ye, Z.; Li, C.; Wan, C.; and Lin, Y. C. 2023. Gpt4aigchip: Towards next-generation ai accelerator design automation via large language models. In *2023 IEEE/ACM International Conference on Computer Aided Design (ICCAD)*, 1–9. IEEE.

Guo, D.; Zhu, Q.; Yang, D.; Xie, Z.; Dong, K.; Zhang, W.; Chen, G.; Bi, X.; Wu, Y.; Li, Y.; et al. 2024. DeepSeek-Coder: When the Large Language Model Meets Programming–The Rise of Code Intelligence. *arXiv preprint arXiv:2401.14196*.

Hong, S.; Zheng, X.; Chen, J.; Cheng, Y.; Wang, J.; Zhang, C.; Wang, Z.; Yau, S. K. S.; Lin, Z.; Zhou, L.; et al. 2023. Metagpt: Meta programming for multi-agent collaborative framework. *arXiv preprint arXiv:2308.00352*.

Jiang, A. Q.; Sablayrolles, A.; Roux, A.; Mensch, A.; Savary, B.; Bamford, C.; Chaplot, D. S.; Casas, D. d. l.; Hanna, E. B.; Bressand, F.; et al. 2024. Mixtral of experts. *arXiv preprint arXiv:2401.04088*.

Johns, D. A.; and Martin, K. 2008. *Analog integrated circuit design*. John Wiley & Sons.

Khan, M. A. M.; Bari, M. S.; Do, X. L.; Wang, W.; Parvez, M. R.; and Joty, S. 2023. xcodeeval: A large scale multilingual multitask benchmark for code understanding, generation, translation and retrieval. *arXiv preprint arXiv:2303.03004*.

Krylov, D.; Khajeh, P.; Ouyang, J.; Reeves, T.; Liu, T.; Ajmal, H.; Aghasi, H.; and Fox, R. 2023. Learning to design analog circuits to meet threshold specifications. In *International Conference on Machine Learning (ICML)*, 17858–17873. PMLR.

Kulal, S.; Pasupat, P.; Chandra, K.; Lee, M.; Padon, O.; Aiken, A.; and Liang, P. S. 2019. Spoc: Search-based pseudocode to code. *Advances in Neural Information Processing Systems (NeurIPS)*, 32.

Li, R.; Zi, Y.; Muennighoff, N.; Kocetkov, D.; Mou, C.; Marone, M.; Akiki, C.; Jia, L.; Chim, J.; Liu, Q.; et al. 2023. StarCoder: may the source be with you! *Transactions on Machine Learning Research (TMLR)*.

Li, Y.; Choi, D.; Chung, J.; Kushman, N.; Schrittwieser, J.; Leblond, R.; Eccles, T.; Keeling, J.; Gimeno, F.; Dal Lago, A.; et al. 2022. Competition-level code generation with alphacode. *Science*, 378(6624): 1092–1097.

Liu, C.; Liu, Y.; Du, Y.; and Du, L. 2024a. LADAC: Large Language Model-driven Auto-Designer for Analog Circuits. *Authorea Preprints*.

Liu, H.; Li, C.; Wu, Q.; and Lee, Y. J. 2024b. Visual instruction tuning. *Advances in neural information processing systems (NeurIPS)*, 36.

Liu, J.; Xia, C. S.; Wang, Y.; and Zhang, L. 2024c. Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation. *Advances in Neural Information Processing Systems (NeurIPS)*, 36.

Liu, M.; Ene, T.-D.; Kirby, R.; Cheng, C.; Pinckney, N.; Liang, R.; Alben, J.; Anand, H.; Banerjee, S.; Bayraktaroglu, I.; et al. 2023a. Chipnemo: Domain-adapted llms for chip design. *arXiv preprint arXiv:2311.00176*.

Liu, M.; Pinckney, N.; Khailany, B.; and Ren, H. 2023b. Verilogeval: Evaluating large language models for verilog code generation. In *2023 IEEE/ACM International Conference on Computer Aided Design (ICCAD)*, 1–8. IEEE.

Liu, S.; Fang, W.; Lu, Y.; Zhang, Q.; Zhang, H.; and Xie, Z. 2023c. Rtlcoder: Outperforming gpt-3.5 in design rtl generation with our open-source dataset and lightweight solution. *arXiv preprint arXiv:2312.08617*.

Lu, Y.; Liu, S.; Zhang, Q.; and Xie, Z. 2024. RTLLM: An open-source benchmark for design rtl generation with large language model. In *2024 29th Asia and South Pacific Design Automation Conference (ASP-DAC)*, 722–727. IEEE.

Luo, Z.; Xu, C.; Zhao, P.; Sun, Q.; Geng, X.; Hu, W.; Tao, C.; Ma, J.; Lin, Q.; and Jiang, D. 2023. Wizardcoder: Empowering code large language models with evol-instruct. *arXiv preprint arXiv:2306.08568*.

Lyu, W.; Yang, F.; Yan, C.; Zhou, D.; and Zeng, X. 2018. Batch Bayesian optimization via multi-objective acquisition ensemble for automated analog circuit design. In *International conference on machine learning (ICML)*, 3306–3314. PMLR.

Olausson, T. X.; Inala, J. P.; Wang, C.; Gao, J.; and Solar-Lezama, A. 2023. Is Self-Repair a Silver Bullet for Code Generation? In *The Twelfth International Conference on Learning Representations (ICLR)*.

Ousterhout, J. K. 1993. An Introduction to TCL and TK.

Pei, Z.; Zhen, H.-L.; Yuan, M.; Huang, Y.; and Yu, B. 2024. BetterV: Controlled Verilog Generation with Discriminative Guidance. *arXiv preprint arXiv:2402.03375*.

Poddar, S.; Budak, A.; Zhao, L.; Hsu, C.-H.; Maji, S.; Zhu, K.; Jia, Y.; and Pan, D. Z. 2024. A Data-Driven Analog Circuit Synthesizer with Automatic Topology Selection and Sizing. In *Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 1–6.

Qin, Y.; Liang, S.; Ye, Y.; Zhu, K.; Yan, L.; Lu, Y.; Lin, Y.; Cong, X.; Tang, X.; Qian, B.; et al. 2023. Toolllm: Facilitating large language models to master 16000+ real-world apis. *arXiv preprint arXiv:2307.16789*.

Razavi, B. 2000a. *Design of Analog CMOS Integrated Circuits*. McGraw-Hill, Inc.

Razavi, B. 2000b. *Design of Analog CMOS Integrated Circuits*. USA: McGraw-Hill, Inc., 1 edition. ISBN 0072380322.

Roziere, B.; Gehring, J.; Gloeckle, F.; Sootla, S.; Gat, I.; Tan, X. E.; Adi, Y.; Liu, J.; Remez, T.; Rapin, J.; et al. 2023. Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950*.

Salvaire, F. 2021. PySpice.

Sparck Jones, K. 1972. A statistical interpretation of term specificity and its application in retrieval. *Journal of documentation*, 28(1): 11–21.

Thakur, S.; Ahmad, B.; Pearce, H.; Tan, B.; Dolan-Gavitt, B.; Karri, R.; and Garg, S. 2023a. Verigen: A large language model for verilog code generation. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*.

Thakur, S.; Blocklove, J.; Pearce, H.; Tan, B.; Garg, S.; and Karri, R. 2023b. Autochip: Automating hdl generation using llm feedback. *arXiv preprint arXiv:2311.04887*.

Thomas, D.; and Moorby, P. 2008. *The Verilog® hardware description language*. Springer Science & Business Media.

Touvron, H.; Martin, L.; Stone, K.; Albert, P.; Almahairi, A.; Babaei, Y.; Bashlykov, N.; Batra, S.; Bhargava, P.; Bhosale, S.; et al. 2023. Llama 2: Open foundation and fine-tuned chat models. *arXiv preprint arXiv:2307.09288*.

Tsai, Y.; Liu, M.; and Ren, H. 2023. Rtlfixer: Automatically fixing rtl syntax errors with large language models. *arXiv preprint arXiv:2311.16543*.

Vladimirescu, A. 1994. *The SPICE book*. John Wiley & Sons, Inc.

Wang, G.; Xie, Y.; Jiang, Y.; Mandlekar, A.; Xiao, C.; Zhu, Y.; Fan, L.; and Anandkumar, A. 2023. Voyager: An Open-Ended Embodied Agent with Large Language Models. In *Intrinsically-Motivated and Open-Ended Learning Workshop@ NeurIPS2023*.

Wang, H.; Wang, K.; Yang, J.; Shen, L.; Sun, N.; Lee, H.-S.; and Han, S. 2020. GCN-RL circuit designer: Transferable transistor sizing with graph neural networks and reinforcement learning. In *2020 57th ACM/IEEE Design Automation Conference (DAC)*, 1–6. IEEE.

Wei, J.; Wang, X.; Schuurmans, D.; Bosma, M.; Xia, F.; Chi, E.; Le, Q. V.; Zhou, D.; et al. 2022. Chain-of-thought prompting elicits reasoning in large language models. *Advances in neural information processing systems (NeurIPS)*, 35: 24824–24837.

Wu, H.; He, Z.; Zhang, X.; Yao, X.; Zheng, S.; Zheng, H.; and Yu, B. 2024. Chateda: A large language model powered autonomous agent for eda. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*.

Yao, X.; Li, H.; Chan, T. H.; Xiao, W.; Yuan, M.; Huang, Y.; Chen, L.; and Yu, B. 2024. HDLdebugger: Streamlining HDL debugging with Large Language Models. *arXiv preprint arXiv:2403.11671*.

Yin, Y.; Wang, Y.; Xu, B.; and Li, P. 2024. ADO-LLM: Analog Design Bayesian Optimization with In-Context Learning of Large Language Models. *arXiv preprint arXiv:2406.18770*.

Zheng, Q.; Xia, X.; Zou, X.; Dong, Y.; Wang, S.; Xue, Y.; Shen, L.; Wang, Z.; Wang, A.; Li, Y.; et al. 2023. Codegeex: A pre-trained model for code generation with multilingual benchmarking on humaneval-x. In *Proceedings of the 29th ACM SIGKDD Conference on Knowledge Discovery and Data Mining (SIGKDD)*, 5673–5684.

Zhong, R.; Du, X.; Kai, S.; Tang, Z.; Xu, S.; Zhen, H.-L.; Hao, J.; Xu, Q.; Yuan, M.; and Yan, J. 2023. LLM4EDA: Emerging Progress in Large Language Models for Electronic Design Automation. *arXiv preprint arXiv:2401.12224*.