A Survey of Circuit Foundation Model: Foundation AI Models for VLSI Circuit Design and EDA

WENJI FANG[†], Hong Kong University of Science and Technology (HKUST), Hong Kong
JING WANG[†], Hong Kong University of Science and Technology (HKUST), Hong Kong
YAO LU, Hong Kong University of Science and Technology (HKUST), Hong Kong
SHANG LIU, Hong Kong University of Science and Technology (HKUST), Hong Kong
YUCHAO WU, Hong Kong University of Science and Technology (HKUST), Hong Kong
YUZHE MA, Hong Kong University of Science and Technology (Guangzhou) (HKUST(GZ)), China
ZHIYAO XIE[*], Hong Kong University of Science and Technology (HKUST), Hong Kong

Artificial intelligence (AI)-driven electronic design automation (EDA) techniques have been extensively explored for VLSI circuit design applications. Most recently, **foundation AI models for circuits** have emerged as a new technology trend. Unlike traditional task-specific AI solutions, these new AI models are developed through two stages: 1) *self-supervised pre-training* on a large amount of unlabeled data to learn intrinsic circuit properties; and 2) *efficient fine-tuning* for specific downstream applications, such as early-stage design quality evaluation, circuit-related context generation, and functional verification. This new paradigm brings many advantages: model generalization, less reliance on labeled circuit data, efficient adaptation to new tasks, and unprecedented generative capability. In this paper, we propose referring to AI models developed with this new paradigm as ***circuit foundation models (CFMs)***. This paper provides a comprehensive survey of the latest progress in circuit foundation models, unprecedentedly covering over 130 relevant works. Over 90% of our introduced works were published in or after 2022, indicating that this emerging research trend has attracted wide attention in a short period. In this survey, we propose to categorize all existing circuit foundation models into two primary types: 1) **encoder-based methods** performing general *circuit representation learning for predictive tasks*; and 2) **decoder-based methods** leveraging *large language models (LLMs) for generative tasks*. For our introduced works, we cover their input modalities, model architecture, pre-training strategies, domain adaptation techniques, and downstream design applications. In addition, this paper discussed the unique properties of *circuits* from the data perspective. These circuit properties have motivated many works in this domain and differentiated them from general AI techniques. Finally, we shared our observed challenges and potential future research directions about developing foundation AI models for EDA methodologies.

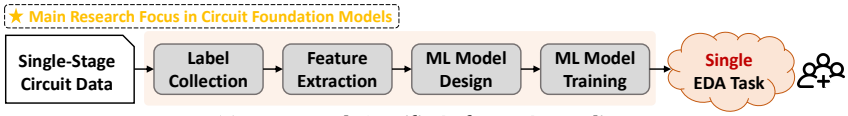
CCS Concepts: • **Hardware → Very large scale integration design**; • **Computing methodologies → Machine learning**.

1 INTRODUCTION

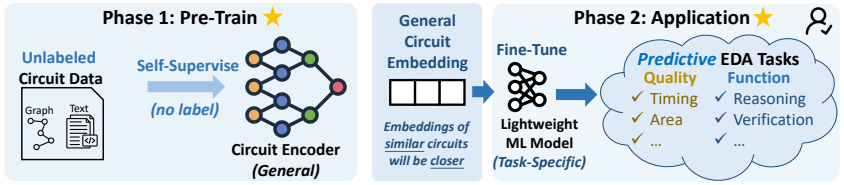
Integrated circuit (IC) is the foundation of our information society. Its complexity has been continuously growing, recently exceeding 100 billion transistors [1]. Such increases in IC complexity have led to sky-rocketing IC design costs, which are estimated to surpass US$500 million for 3nm technology [2]. These challenges result in a compelling need to improve IC design efficiency, possibly achieved by ground-breaking next-generation electronic design automation (EDA) techniques. Many EDA practitioners in both academia and industry have placed high hopes in new artificial intelligence (AI) or machine learning (ML) methods in IC design and EDA techniques, targeting more agile design for lower IC *design costs*, less *human efforts*, and shorter *turnaround time*.

AI for EDA/chip design. In recent years, AI for chip design, also named AI/ML for EDA or AI-assisted EDA [3, 4], has been viewed as a highly promising technique, owing to its ability to *reuse knowledge* from prior circuit design data. Relevant AI-driven EDA techniques are also adopted in commercial EDA tools [5, 6]. Various ML models can be trained to provide early predictions or

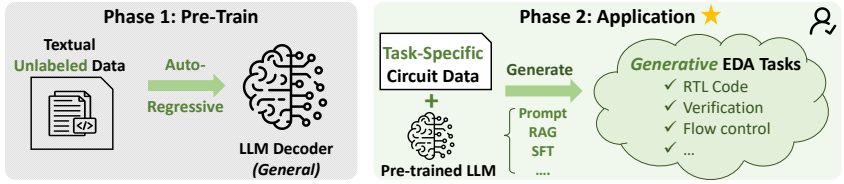
[†]These authors contributed equally to this work.
[*]Corresponding author (eezhiyao@ust.hk).



(a) Type I: Task-Specific AI for EDA Paradigm



(b) Type II: General Encoder-Based Circuit Foundation Model Paradigm



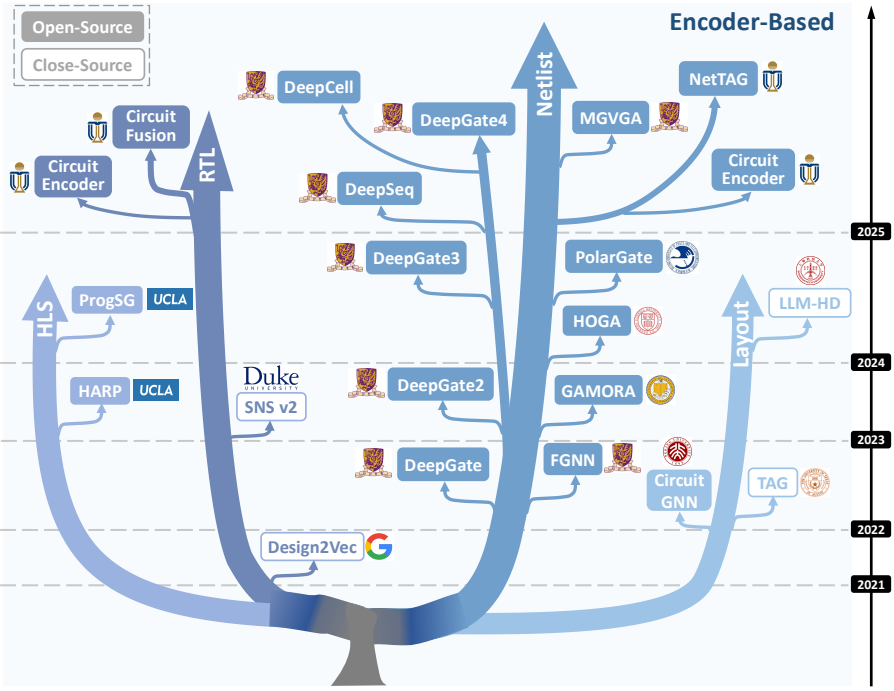
(c) Type II: General Decoder-Based Circuit Foundation Model Paradigm

Fig. 1. Different paradigms of AI for EDA techniques. (a) Type I: Supervised Predictive AI Techniques for EDA. This type of work has been extensively studied. **(b) & (c) Type II: Foundation AI Techniques for EDA (i.e., Circuit Foundation Models)**. This type of work includes two paradigms, named encoder-based and decoder-based circuit models. Both paradigms develop the foundation AI model through two stages: self-supervised *pre-training* and *fine-tuning*. Our survey will focus on the emerging type II methods.

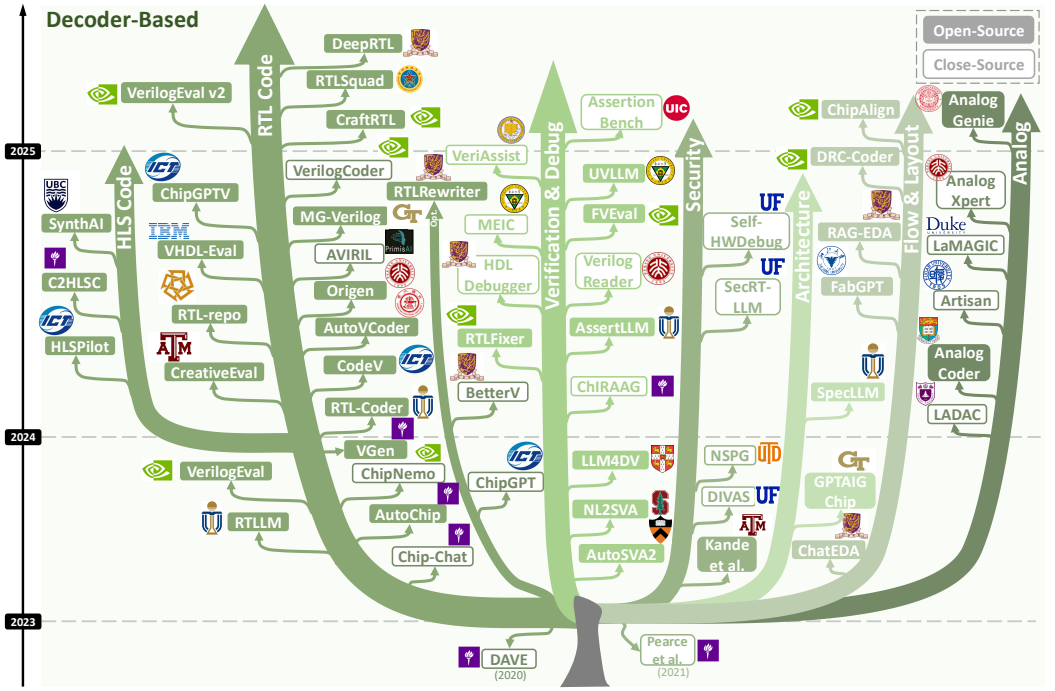
optimizations for circuits, bypassing time-consuming downstream design and simulation steps. Learning from prior design solutions, ML models can perform circuit quality evaluations at early design stages and thus guide early design optimizations. Existing AI for EDA techniques have been extensively explored for almost all standard VLSI design stages (e.g., architecture stage, high-level synthesis (HLS) code, register-transfer level (RTL) code, gate-level netlist, post-placement layout, clock tree, and post-routing layout) and all primary circuit design objectives (e.g., timing, power, area, congestion, IR drop, signal integrity, and functionality).

Foundation AI for EDA/chip design: a new trend and our focus. Recently, general foundation AI models in natural language processing (NLP) and computer vision (CV) (e.g., BERT [7], CLIP [8], DALLE [9], and ChatGPT [10]) have emerged and represent a significant leap in AI techniques. These foundation models, characterized by their large model scale and application scopes, have demonstrated an incredible ability to understand, predict, and generate content [11]. In comparison with these foundation models in NLP and CV, previous AI applications in *circuits* lag far behind well-explored general natural languages and images. This has motivated the latest trend of exploring foundation AI models for EDA techniques and circuit design applications.

The trending works on foundation AI for EDA have demonstrated unprecedented ability in model generalization, few-shot learning, and generation tasks. These models typically leverage a two-stage paradigm of *pre-training* on large-scale datasets followed by *fine-tuning* for specific applications, significantly enhancing adaptability across various EDA tasks. Their great potential has attracted wide attention from the EDA community. Some representative works [12–15] are relatively highly cited since their publication, compared with average EDA publications. However, there is a lack of systematic definition, analysis, or survey on this series of latest works, leading to confusion when discussing many concepts in our communities (e.g., large circuit model vs. LLM-aided design vs. AI agents for EDA). In this survey paper, we will cover all representative



(a) *Encoder-based* circuit foundation model, covered in Section 4.



(b) *Decoder-based* circuit foundation model, covered in Section 5.

Fig. 2. Evolutionary tree of foundation AI models for VLSI design and EDA.

works on foundation AI for EDA. We propose referring to this type of work as **circuit foundation models (CFMs)**. Figure 2 illustrates the evolutionary tree of existing circuit foundation models, including both *encoder-based* and *decoder-based* paradigms. This paper also covers the potential and challenges of CFMs from our perspective.

Structure of Section 1. In this Introduction, we will first propose our own taxonomy of existing AI for EDA techniques in Section 1.1, categorizing all existing AI for EDA techniques into two major types. Then we will briefly introduce the already extensively studied Type I techniques (supervised AI for EDA) in Section 1.2 and elaborate on the emerging Type II techniques (foundation AI for EDA, the focus of our survey) in Section 1.3. After that, in Section 1.4, we will summarize all existing surveys that cover similar topics and elaborate on the contributions of this survey. In Section 1.5, we will introduce the overall structure of this whole survey paper.

1.1 Our Taxonomy of AI for EDA Techniques: Two Different Types

In this survey, we propose to categorize existing AI for EDA techniques into two main types, as listed below. Figure 1 summarizes and compares all three paradigms of these two types of works.

- **Type I: Supervised Predictive AI Techniques for EDA.** The mainstream paradigm of previous AI for EDA solutions adopts *supervised predictive* AI models. These supervised predictive models have been developed for various applications, including early-stage design quality prediction, fast design quality simulation, design space exploration, etc. Relevant works have been extensively studied and covered in existing surveys [3, 4] and book [16].
- **Type II: Foundation AI Techniques for EDA (*Circuit Foundation Model*).** This trending technique is the focus of this survey. The development of foundation AI solutions, according to our proposed definition, involves two phases: 1) Pre-training phase; 2) Fine-tuning phase. The first *pre-training* step, which is typically *self-supervised* on a large amount of unlabeled data, enables the AI model to learn more general circuit intrinsic patterns. The subsequent *fine-tuning* step can efficiently make the model adapt to specific EDA tasks.

  Figure 1(b) & (c) summarize two different paradigms of foundation AI models for circuits. We propose to incorporate both paradigms into the scope of *circuit foundation models*:
  - **Encoder-based circuit foundation models.** One primary paradigm performs *circuit representation learning* to support *predictive tasks*. They typically encode a circuit design into a general embedding (i.e., a vector with rich circuit information). This embedding will be the input to lightweight downstream models for various EDA applications.
  - **Decoder-based circuit foundation models.** The other primary paradigm performs decoding tasks, thus supporting *generative tasks*. They typically adopt decoder-based large language models (LLMs) to help *generate* circuits, including design HLS or RTL code, design functionality descriptions, verification assertions, EDA tool scripts, etc.

1.2 Type I: Supervised Predictive AI Techniques for EDA (covered in prior surveys)

Existing AI for EDA methods are mostly tailored to specific tasks, such as early prediction of various design quality metrics (e.g, timing [17–22], area [23–26], power [27–35], IR drop [36–40], routability [41–47], crosstalk [48–50], and manufacturability [51–53]) or the reasoning of circuit functionalities [54–58] for verification applications. Additionally, tasks for circuit optimization (e.g., flow tuning [59–61], design space exploration [62–64], design quality optimization [65, 66]) also largely rely on the prediction of circuit quality to provide feedback. As Figure 1 (a) shows, these methods are typically developed by supervised training, which requires extensive label collection,

model customization, and model development for every single task. Despite obvious effectiveness, this mainstream supervised paradigm has several inter-related general limitations:

(1) **Difficulty to get sufficient labeled data.** It is typically difficult to accumulate sufficient labeled training data: 1) Many coarse-grained prediction tasks do not support many labels. For example, to predict the layout area of a netlist, each circuit layout only provides one label (i.e., its layout area). 2) The label generation process is inherently highly time-consuming. A dilemma is, most predictive AI models are trained to bypass the the slowest design/simulation steps. However, these slowest steps are exactly required to collect labels.

(2) **Time-consuming AI model development process.** The development process of supervised task-specific solutions is tedious and time-consuming. The development steps include circuit collection, label generation, feature engineering, model architecture design, model training, and model testing. This whole process easily takes months of engineering efforts.

(3) **Lack of generalization across tasks.** Since supervised task-specific models cannot be directly generalized to other tasks, it leads to an inefficient repetitive development of ML solutions. Moreover, from the methodology perspective, it implies that these supervised ML solutions only *learned* task-specific patterns, instead of *understanding* more general knowledge of target circuit designs.

Due to the page limit and the large number of extensively explored type I works, we will not exhaustively cover all prior type I works. For a more comprehensive list of type I supervised predictive works, we refer our readers to prior surveys [3, 4] and a book [16] co-authored by many researchers in this domain.

1.3 Type II: Foundation AI Techniques for EDA (the focus of this paper)

This survey focuses on the emerging paradigms of foundation AI techniques for EDA. As illustrated in Figure 1 (b) and (c), this type of technique leverages pre-trained foundation AI models for circuits (referred to as *circuit foundation models*), which can be efficiently fine-tuned using a small amount of task-specific labeled circuit data. Compared to traditional task-specific supervised AI for EDA solutions, this type II techniques offer significant advantages:

(1) **Learning unlabeled circuit intrinsics.** Circuit foundation models are typically pre-trained on a large amount of unlabeled data, enabling them to capture the underlying intrinsic information about circuits, without requiring expensive labeled datasets.

(2) **Efficient fine-tuning for solving EDA task.** Well-pre-trained models require only a small amount of labeled data for fine-tuning. It significantly reduces the time and resources needed to solve each specific EDA task compared to training models from scratch.

(3) **Generalization across various tasks.** General circuit intrinsics learned by foundation models can be adapted to multiple tasks, making the models versatile and reducing the need for repetitive task-specific model development.

(4) **Unprecedented generative capability for EDA tasks.** Some circuit foundation models exhibit remarkable generative capabilities, unprecedentedly automating tasks such as circuit code generation, assertion generation for verification, and design flow script generation. These models go beyond existing predictive tasks, enabling innovative AI-driven solutions that enhance design productivity and streamline circuit development flow.

Encoder-based circuit foundation model. Figure 1 (b) demonstrates the paradigm of circuit encoders. Circuit encoders transform various circuit modalities (e.g., graphs or text formats) into generalized embeddings that contain rich intrinsic circuit properties. These encoders are typically *pre-trained on circuit data*. Due to the uniqueness of the circuit data compared with well-studied

images or natural language, encoder models have to be specifically customized to handle circuit data. Research works primarily focus on two aspects: (1) in phase 1, developing specialized ML architectures and pre-training techniques to effectively capture circuit semantics, structural information, and physical attributes, and (2) in phase 2, leveraging pre-trained circuit encoders to support various predictive EDA tasks, including design quality evaluation and functional reasoning. In this survey, we systematically categorize existing circuit encoders according to their respective design stages and provide a comprehensive analysis of their supported downstream tasks.

Decoder-based circuit foundation model. Figure 1 (c) illustrates the paradigm of circuit decoders. Circuit decoders typically leverage LLMs as their backbone, which are typically *extensively pre-trained on vast text datasets* spanning multiple domains. Leveraging the powerful pre-trained LLMs, circuit decoders mainly focus on domain adaptation to circuit-related generative tasks, such as prompt engineering, fine-tuning, retrieval-augmented generation, etc. In this survey, we categorize existing decoder-based methods based on their application domains, covering key areas such as circuit code generation, verification, design flow automation, etc. For each category, we analyze representative benchmarks, model development techniques, and the latest advancements.

Key differences between encoder- and decoder-based models are summarized below:

(1) **Circuit modality as input**: Encoders primarily process *graph-based* circuit structures, such as netlists and control-data flow graphs, often leveraging graph learning models. Some recent works integrate multimodal learning, combining structural graphs with textual descriptions. In contrast, decoders focus on *text-based* formats like HDL code and natural language specifications, utilizing LLMs for interpretation and generation.

(2) **Circuit learning techniques**: Encoders require customized pre-training and fine-tuning on circuit data. They are typically built from scratch using graph AI models. There is no standard architecture for circuit encoding, leading to diverse model designs and self-supervised learning techniques. In contrast, decoders typically rely on LLMs already extensively pre-trained on vast text datasets. Relevant works rely on existing pre-trained LLMs in the public domain, including both open-sourced (e.g., Llama, Mistral, DeepSeek) and commercial (e.g., GPT-3.5, GPT-4o) LLMs. These works focus on adaptation to the circuit domain through prompt engineering, fine-tuning, and retrieval augmented generation (RAG).

(3) **Target downstream tasks**: Encoders typically support predictive tasks such as design quality evaluation and functional reasoning, leveraging encoded circuit embeddings. Decoders are typically tailored for generative tasks, such as circuit code generation, verification automation, design flow generation, etc.

1.4 Comparison of Existing Relevant Surveys and This Paper.

Table 1 compares all existing survey papers [11, 67–73] about foundation AI models for circuit applications. Notably, almost all surveys [67–73] focus only on decoder-based models (i.e., LLM for EDA). This trend reflects the rapid evolution of LLMs and their significant potential for generative EDA tasks, such as HDL code generation, verification, debugging, etc. Among these surveys on decoder-based LLMs for EDA, some surveys [67–69] try to provide comprehensive reviews on multiple relevant tasks, while some others [70–73] focus on one specific topic, mostly about *circuit security*. The only exception is a special perspective paper [11] co-authored by many EDA researchers. It advocates for an ambitious framework of multiple *encoder*-based foundation models aligned across design stages. This envisioned concept is named large circuit model (LCM) [11]. Different from existing surveys, our survey paper incorporates both encoder-based and decoder-based circuit foundation models, analyzing their similarities and differences.

Surveys	Design Generation	Design Verification	Design Debugging	Design Security	Design Optim.	Design Flow	Encoder -based	Decoder -based	Time Published	No. of CFM Works Covered
[70]				✓				✓	2023-10	15
[67]	✓	✓	✓		✓			✓	2023-12	22
[11]	✓					✓	✓	✓	2024-03	6 (Encoder) + 21 (Decoder)
[71]	✓		✓	✓				✓	2024-04	14
[72]	✓			✓	✓	✓		✓	2024-05	32
[73]	✓	✓	✓	✓				✓	2024-06	24
[74]	✓	✓	✓			✓		✓	2024-10	29
[69]	✓	✓	✓					✓	2024-12	71
[68]	✓					✓		✓	2025-01	39
Ours	✓	✓	✓	✓	✓	✓	✓	✓	2025-03	21 (Encoder) + 111 (Decoder)

Table 1. Comparison of *existing surveys* on foundation models for chip design, covered in Section 1.4.

Table 1 also reports the number of CFM-related works covered in each survey paper. We only count works in the scope of the circuit foundation model (i.e., pre-train and fine-tuned AI models targeting circuit design tasks). Partially due to the fast development in this emerging direction, most existing surveys only covered less than 40 related works. In comparison, our comprehensive survey unprecedentedly introduces the largest number of (i.e., over 130) relevant works, covering all key circuit design tasks listed in Table 1. We briefly introduce each existing survey and highlight the unique contributions of our study below.

Surveys on decoder models covering broad tasks. *LLM4EDA* [67] is an early *comprehensive review*, covering various EDA tasks such as chatbot-based methods, circuit code and script generation, and circuit verification. However, since it was published in 2023 and this direction developed very fast, it only covered 22 works. *Xu et al.* [74] summarize 29 early-stage studies on circuit code generation, debugging, verification, and physical implementation. While it provides insights into these areas, it lacks coverage of the latest developments and broader topics such as security, design optimization, and architecture design. *Abdollahi et al.* [69] provide a more *extensive* survey, analyzing 71 studies on LLM-assisted circuit design, including applications in circuit generation, verification, and debugging. However, possibly due to their automated literature screening process, we observed several incorrect descriptions in this survey. For example, the survey [69] incorrectly categorizes works of [75–77] as LLM-aided design methodologies, while these works actually primarily focus on the acceleration of LLMs (i.e., designing hardware accelerators). A recent survey by *Pan et al.* [68] reviews LLM applications in EDA. Despite its recency, it still only covers a limited number of works (39 works), primarily focusing on design generation and design flow automation. It lacks a broader discussion on design verification, security, architecture design, and analog tasks.

Surveys on decoder models covering a specific task. In addition to surveys targeting broad EDA applications, the other series of surveys [70–73] focus specifically on *circuit security topics* with LLM-assisted techniques. *Saha et al.* [70] pioneered the discussion of integrating LLMs into the SoC security verification paradigm in 2023. At the time, not many specialized LLM-based solutions had been customized for SoC security. Therefore this work [70] primarily summarizes applying *general LLM techniques* in hardware security tasks. In 2024, three short surveys (all less than 7 pages) [71–73] cover LLM methods for hardware security, each covering 10 to 30 works.

A perspective paper on LCM. In 2024, a special perspective paper [11] co-authored by many EDA researchers proposes and advocates an interesting and ambitious concept of *large circuit model (LCM)*. This LCM can be viewed as an envisioned framework of multiple aligned encoder-based circuit foundation models, each devoted to one design stage. This paper also reviews both supervised task-specific AI solutions and foundation AI models for EDA (i.e., 6 encoders and 21

decoders). It identifies key challenges in developing large-scale circuit encoders and sets the stage for future advancements in encoder-based circuit foundation models.

The contributions of this survey, compared with prior surveys, can be summarized below:

(1) This survey proposes the concept of *circuit foundation model*. It incorporates both encoder-based circuit representation learning techniques and decoder-based LLM for EDA methods into a unified framework, enabling comparison between these two paradigms.

(2) This comprehensive survey systematically introduces over 130 works. All existing circuit foundation models cited in prior surveys [11, 67–73] have been covered in this work.

(3) The 21 encoder-based models span all standard design stages, including HLS, RTL, netlist, and layout stages, along with their supporting predictive EDA tasks.

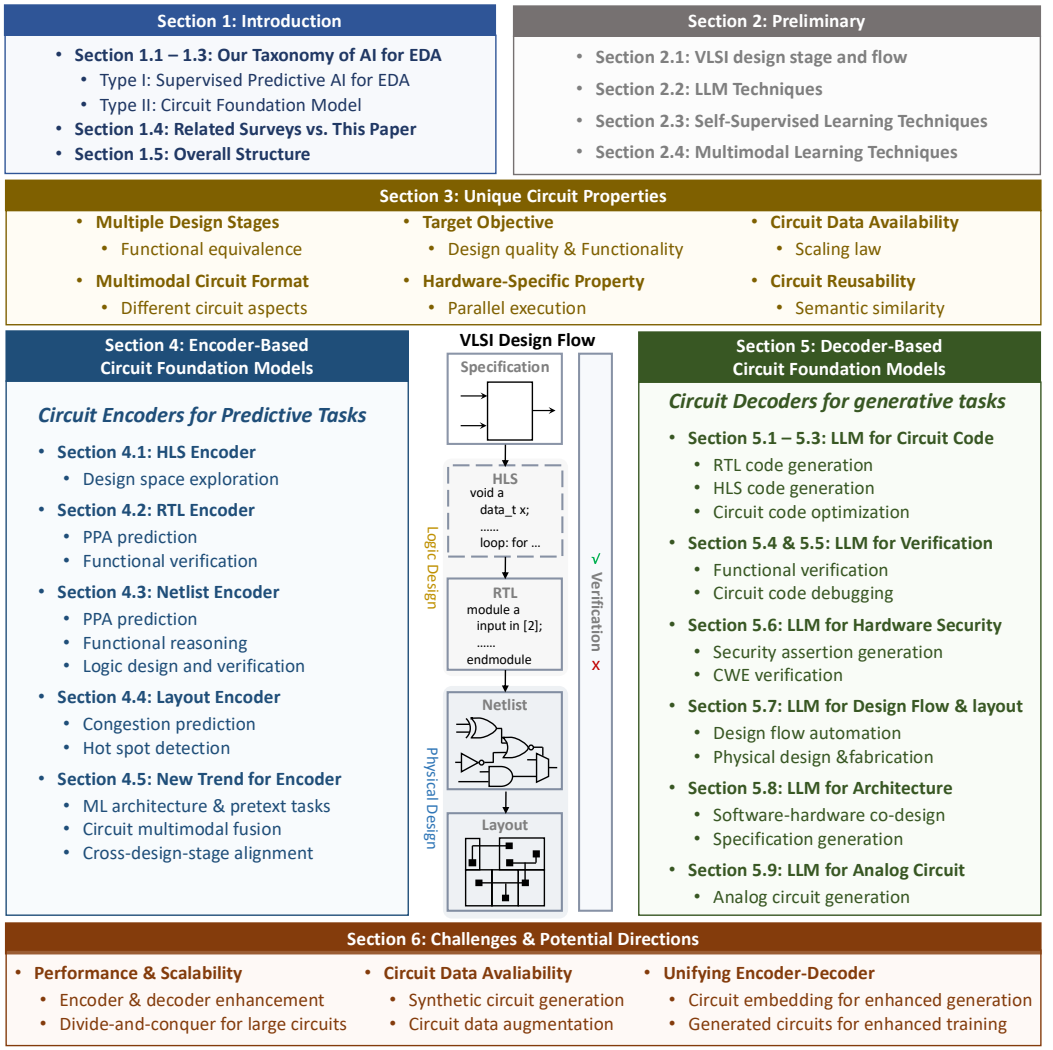


Fig. 3. Overview of this survey paper. Section 2 provides the background of VLSI circuit design and foundation AI model techniques. Section 3 discusses the unique properties of circuit data that motivate AI-driven solutions. Section 4 and Section 5 comprehensively review existing circuit encoders and decoders, respectively. Finally, Section 6 explores key challenges and future directions in circuit foundation models.

(4) The 111 decoders-based models cover all mainstream EDA applications, including VLSI circuit code processing (generation, optimization, verification, and debugging), hardware security, design flow automation, physical design, architecture design, and analog design.
(5) Besides the in-depth analysis of these approaches, we highlight key advancements, challenges, and future research directions to further enhance circuit foundation models and their impact on modern VLSI design automation.

This survey tries to cover all publications within the scope of the circuit foundation model, including journals, transactions, conference and workshop proceedings, thesis, and pre-prints. However, very short articles (e.g., late-breaking results, experiment reports) that are equal to or less than 3 pages may not be covered. For the same work with multiple versions and possibly different titles, we will avoid duplicated citations and tend to *cite the latest version*. When counting the publication date, we use the *date when the **earliest** version* gets released to the public.

1.5 Overall Structure of This Survey Paper

Figure 3 provides the overall structure of this paper.

- In Section 2, we will summarize related **preliminary knowledge**, covering both standard VLSI circuit design flow (Section 2.1) and basic techniques of general foundation AI models, including LLM techniques (Section 2.2), self-supervised learning techniques (Section 2.3), multimodal learning techniques (Section 2.4).
- In Section 3, we will introduce all our observed **unique properties of circuit data**. These properties have largely motivated many CFM works in this survey, and differentiate these works from general AI solutions in other domains (e.g., CV, NLP).
- In Section 4, we will cover all existing **encoder-based circuit foundation models**, covering the HLS stage (Section 4.1), RTL stage (Section 4.2), netlist stage (Section 4.3) and layout stage (Section 4.4). The emerging and more advanced circuit encoder techniques will be covered in Section 4.5.
- In Section 5, we will cover all existing **decoder-based circuit foundation models**, covering all application domains: RTL code generation (Section 5.1), HLS code generation (Section 5.2), design optimization (Section 5.3), hardware code verification (Section 5.4), hardware code debugging (Section 5.5), hardware design security (Section 5.6), design flow automation and layout design (Section 5.7), hardware architecture design (Section 5.8), and analog circuit design (Section 5.9).
- In Section 6, we will analyze the **challenges and opportunities** of the circuit foundation models, based on our own research experience.

2 PRELIMINARY

Before covering specific CFM works, in this Section, we first summarize **preliminary knowledge** related to circuit foundation model, covering both standard VLSI circuit design flow in Section 2.1 and the basic techniques of foundation AI models, including LLM techniques in Section 2.2, self-supervised learning techniques in Section 2.3, multimodal learning techniques in Section 2.4.

2.1 Standard VLSI Design Stage and Flow

A standard VLSI circuit design flow comprises several stages: specification definition, RTL design, logic synthesis, and physical design, as shown in the center of Figure 3. At each stage, the design is represented in the corresponding format: specification, RTL code, netlist, and layout. In addition to these standard stages, high-level synthesis (HLS) is sometimes employed for more agile design or FPGA prototyping, based on HLS code in C/C++/SystemC. Verification and design quality analysis are carried out at various stages to ensure functional correctness and meet design quality constraints, respectively. Together, these stages transform the initial design specifications into a manufacturable and verified digital circuit layout. We introduce each design stage below.

Specification definition. The design process begins with a clear natural language specification that outlines the expected functionality, as well as performance, power, and area (PPA) requirements for a target digital circuit. This specification serves as the blueprint for subsequent design steps.

HLS code design. The specification can be translated into an abstract design using high-level programming languages or description languages like C/C++ or SystemC. Designers develop *algorithms* that meet the functional requirements. The algorithms are described at a high level, focusing on functionality rather than hardware specifics.

RTL design. RTL design is the process of translating the high-level specification into a more detailed and implementable representation using hardware description languages (HDLs) such as Verilog or VHDL. These HDLs describe the behavior of digital circuits at the register-transfer level. The HDL code captures how data moves between registers (i.e., sequential registers) and how logic gates operate on that data within each clock cycle (i.e., combinational logic). Viewing each design as a finite-state machine, RTL defines the state transitions across clock cycles, ensuring that the circuit responds correctly to changes in input signals and synchronizes with the clock.

Logic synthesis. Logic synthesis converts high-level RTL designs into low-level, optimized gate-level netlists. This process consists of three key steps: translation (i.e., elaboration), optimization, and technology mapping. First, the synthesis process begins by translating the RTL code into an intermediate representation, such as the AND-Inverter Graph (AIG) in synthesis tools like ABC [78]. The synthesis tool then optimizes the logic based on constraints like delay and logic depth. Finally, technology mapping is performed, where the optimized logic is mapped to specific gates from a technology library provided by semiconductor foundries. This library contains various gate types, each with unique characteristics. The final output is a gate-level netlist, which represents the circuit in terms of logic gates and their interconnections.

Physical design. Physical design translates the gate-level netlist into a manufacturable physical layout. This process includes several key steps: floor planning, placement, clock tree synthesis (CTS), and routing. The first step, floor planning, involves arranging the major functional blocks of the chip in a way that optimizes performance while minimizing area. Designers determine the approximate locations of various components. Following floor planning, placement positions individual gates and components within the predefined floorplan, aiming to minimize wire length and ensure efficient placement. CTS follows placement, where the clock distribution network is designed to ensure the proper synchronization of all clock signals across the chip. Finally, routing connects the placed components using metal layers to form the required electrical connections.

During routing, considerations such as signal integrity, minimization of crosstalk, and adherence to design rules are essential to ensure the layout is functionally correct and manufacturable.

Verification. Verification ensures the design meets specifications [79] and includes functional and physical verification. Functional verification checks if the design meets its specifications, using testbench simulations to model real-world conditions. Formal verification applies mathematical techniques, with equivalence checking to ensure consistency between design representations (e.g., RTL and gate-level). Physical verification ensures the layout complies with manufacturing constraints using design rule checking (DRC) and layout versus schematic (LVS) to detect and correct violations for manufacturability.

Analysis. Analysis evaluates the design against performance metrics to ensure it meets specifications. Static timing analysis (STA) verifies that timing constraints are met, ensuring signals propagate within required time limits. Power analysis estimates both dynamic (switching activity) and static (leakage currents) power consumption. Signal integrity analysis checks for issues like crosstalk, noise, and electromagnetic interference. Additionally, thermal analysis assesses heat generation and dissipation to ensure proper thermal management and reliable operation.

2.2 LLM Techniques in AI Foundation Models

The evolution of LLMs marks a pivotal advancement in artificial intelligence, particularly in NLP. Before delving into their applications in circuit design, it's essential to understand their techniques. Below, we introduce the brief evolution history of LLM and the key techniques employed in the morden advanced LLM models. This foundational understanding highlights the transformative potential of LLMs across various domains, including circuit design.

A brief history of LLM. LLMs have evolved from early rule-based approaches to modern deep learning-driven foundation models, significantly advancing natural language processing (NLP). These advancements have enabled models like BERT and GPT to capture complex semantic and contextual nuances, leading to breakthroughs in various language-related tasks. Below, we summarize the key evolutionary stages of LLM development.

(1) **Rule-based method:** The earliest NLP systems relied on manually crafted linguistic rules and statistical models [80]. These approaches defined explicit syntactic and semantic rules for processing text but were limited in scalability and adaptability. While rule-based methods could handle predefined patterns effectively, they struggled with the complexity and variability of natural language, making them unsuitable for large-scale applications.
(2) **ML solution by manual feature engineering:** The introduction of statistical machine learning improved NLP by enabling data-driven language modeling. Early machine learning solutions required extensive manual feature engineering, where domain experts designed handcrafted features such as n-grams, part-of-speech tags, and dependency structures. Traditional models, including Hidden Markov Models (HMMs) and Support Vector Machines (SVMs), demonstrated better adaptability than rule-based methods but still relied on human-designed representations, limiting their generalization capabilities.
(3) **Task-specific deep learning:** The emergence of deep learning revolutionized NLP by replacing manual feature engineering with automatic representation learning. Models like Word2Vec [81] and GloVe [82] introduced word embeddings, representing words in continuous vector spaces to capture semantic relationships. Recurrent Neural Networks (RNNs)[83] and Long Short-Term Memory Networks (LSTMs)[84] further improved sequence modeling by capturing contextual dependencies in text. However, these models faced challenges with long-range dependencies and computational efficiency due to their sequential nature.

(4) **General transformer-based foundation model:** The introduction of the Transformer architecture marked a paradigm shift in NLP. Transformers utilize self-attention mechanisms to process entire sequences in parallel, capturing global dependencies efficiently. Encoder-based models like BERT excel in understanding context through bidirectional masked language modeling, while decoder-based models like GPT specialize in generative tasks using autoregressive token prediction. These foundation models are pre-trained on massive datasets and fine-tuned for various downstream applications, eliminating the need for task-specific model development. Their success has extended beyond NLP, inspiring new research directions in domains such as circuit design, where they are increasingly being used for tasks such as RTL code generation, verification, and design optimization.

Key techniques in decoder-only LLMs. Modern decoder-only LLMs, such as GPT, leverage a range of advanced techniques to enhance their performance and adaptability across various tasks. Below, we summarize five key techniques used in state-of-the-art decoder-based language models. **(1) Auto-regressive generation:** Decoder-only LLMs follow an auto-regressive approach, where they generate text sequentially, predicting one token at a time based on previously generated tokens. This autoregressive process allows models to produce coherent and contextually relevant text, making them highly effective for generative tasks such as text completion, summarization, and code generation. **(2) Prompt engineering:** Prompt engineering involves carefully crafting input text (prompts) to guide LLMs toward producing desired outputs. Since decoder-based models lack inherent task-specific fine-tuning for every possible use case, effective prompting helps steer model behavior without requiring additional training. Techniques such as zero-shot prompting (providing a task description), few-shot prompting (including examples), and chain-of-thought prompting (explicit reasoning steps) have been widely explored to enhance model performance across different applications. **(3) Supervised fine-tuning (SFT):** SFT refines pre-trained LLMs on specific datasets with labeled examples, enabling better adaptation to specialized tasks. By providing high-quality training examples, SFT improves accuracy and reliability in domain-specific applications, such as HDL code generation, circuit verification, and design optimization. Many domain-adapted LLMs, including those for EDA tasks, leverage SFT to improve performance on structured data and technical domains. **(4) Retrieval-augmented generation (RAG):** RAG enhances LLMs by incorporating external knowledge sources during inference. Instead of relying solely on pre-trained knowledge, the model retrieves relevant documents or contextual information from databases, augmenting its response with up-to-date and factual content. This technique is particularly useful for knowledge-intensive applications, such as circuit debugging and design flow optimization, where dynamic information retrieval improves response accuracy and relevance. **(5) Reinforcement learning from human feedback (RLHF):** RLHF refines LLM behavior using human preferences to optimize response quality. In this approach, human annotators rank model outputs, and reinforcement learning algorithms adjust the model's reward function to align responses with human expectations. RLHF has been instrumental in making LLMs more aligned with human intent, improving coherence, factual correctness, and ethical considerations in generated outputs.

2.3 Self-Supervised Learning Techniques in AI Foundation Model

In the development of AI foundation models, a variety of machine learning techniques are employed to enable these models to generalize across a wide range of tasks. These techniques are categorized into two primary phases: self-supervised learning for pre-training and supervised fine-tuning for downstream tasks.

Self-supervised learning is a powerful technique that allows models to learn from unlabeled data by generating labels from the data itself, often using auxiliary tasks. This phase helps the model understand general representations that can later be fine-tuned for specific tasks. We demonstrate the representative self-supervised learning techniques below.

- **Contrastive learning**. This method learns representations by comparing similar (positive) and dissimilar (negative) pairs. For example, in image processing, positive pairs may be different augmentations of the same image, while negative pairs come from different classes. This method helps models generate useful embeddings for downstream tasks like retrieval or classification. It has been successful in computer vision and natural language processing (e.g., SimCLR [85], CLIP [8], MoCo [86]).
- **Mask-reconstruction.** This method involves randomly masking parts of the input and training the model to predict the missing information. This forces the model to learn context from the surrounding data, improving its ability to understand structure and relationships. In NLP, BERT [7] predicts masked words in sentences, while in vision tasks, Masked Autoencoders (MAE) [87] show how masking portions of an image can lead to effective representation learning, aiding tasks like classification or segmentation.
- **Auto-regressive.** The auto-regressive method involves predicting the next element in a sequence, given the previous elements. In natural language processing, models like GPT [88] generate coherent text by predicting the next word based on the preceding context. In vision tasks, pixel-based auto-regressive models predict pixel values given prior pixels, such as in PixelCNN [89]. The strength of auto-regressive models lies in their ability to learn complex dependencies within sequential or spatial data, allowing them to generate high-quality outputs for tasks like text generation, image synthesis, and beyond.

After pre-training with self-supervised methods, foundation AI models are fine-tuned on labeled data to adapt to specific tasks. This fine-tuning process enhances their performance across various applications, such as in the domain of NLP and CV.

2.4 Multimodal Learning Techniques in AI Foundation Model

Multimodal learning techniques are essential for AI foundation models, as they enable the integration and processing of multiple data modalities such as text, images, and video. We summarize the key multimodal learning techniques into two categories: multimodal encoders for representation learning and multimodal decoders for generation.

Multimodal encoders for representation learning focus on learning joint representations across multiple modalities, allowing the model to extract and relate information effectively. Notable examples include CLIP [8], which aligns visual and textual representations to enable zero-shot learning for tasks like image classification and retrieval. ALBEF [90] builds upon CLIP by aligning text and image representations and then performing multimodal fusion, improving performance in multimodal reasoning tasks such as visual question answering. These techniques lay the foundation for multimodal circuit representation learning, where textual descriptions (e.g., HDL code), structural graphs (e.g., netlists), and layout images can be effectively integrated for comprehensive circuit analysis and optimization.

Multimodal decoders for generation utilize one modality as input to generate content in another modality, such as describing images with text or synthesizing images and videos from textual descriptions. The BLIP family [91, 92] bridges image understanding and text generation by introducing a connector that adapts image embeddings for frozen LLMs, enabling accurate textual descriptions of images. LLaVA [93] enhances image understanding by fine-tuning LLMs with visual-text instruction pairs, improving the model's ability to process and describe images. Extending this

approach to video, Video-LLaVA [94] generates video content based on textual or image-based inputs. Beyond generating text from visual inputs, some models focus on the reverse task—creating visual content from textual descriptions. DALL·E [95] pioneers this field by generating diverse and high-quality images from textual prompts, facilitating creative content synthesis. Parti [96] further refines this capability, enabling the generation of high-resolution, contextually accurate images from detailed prompts. These advancements in multimodal generation highlight the potential of circuit foundation models, where similar approaches could be employed to generate circuit layouts from textual specifications, convert hardware design schematics into structured descriptions, or facilitate design debugging by linking textual analysis with circuit visualizations.

3 UNIQUE CIRCUIT DATA PROPERTIES

In this section, we will summarize the unique properties of circuits, especially from the data perspective. We will compare the circuit data with other common data formats, such as general images or natural languages. Understanding these unique properties of circuits is important, since they largely motivate many circuit foundation models and thus differentiate these CFM from general AI solutions in other domains like CV or NLP.

Equivalence across design stages. In the standard digital IC design flow, which includes specification, HLS code, RTL, netlist, and layout, ensuring equivalence across these stages is crucial for maintaining the integrity of the design. Each design stage refines the design from an abstract specification into a more detailed representation, but the underlying functionality and performance must remain consistent. This concept of equivalence has led to the use of circuit equivalent transformations as a data augmentation technique, allowing for the generation of multiple, functionally equivalent representations of a circuit. Furthermore, it has inspired cross-design-stage alignment in circuit foundation models, enabling these models to capture and align information across different stages of the design flow. This alignment enhances the model's ability to transfer knowledge between stages and improves cross-stage consistency.

Multimodal circuit format. As shown in Figure 4, circuit data inherently can be represented in multiple formats and modalities, each capturing different aspects of the circuit, including:

- **Text.** This modality includes hardware description languages (HDLs) such as Verilog and VHDL, along with high-level specifications in natural language. Text-based representations define circuit functionality, behavioral constraints, and design requirements, emphasizing semantic information of circuits.
- **Graph.** Circuit structures are naturally represented as graphs, where nodes correspond to components (e.g., logic gates, registers, functional blocks) and edges capture connectivity (e.g., data flow, control dependencies). Graph-based formats, including control-data flow graphs and gate-connected graphs, preserve the topological relationships, which are crucial for structural reasoning in EDA tasks.
- **Image.** The physical layout of circuits, particularly at the post-synthesis stage, can be represented in two-dimensional visual formats, similar to the format of images. These 'images' capture geometric features, including component placement and interconnect routing, which are critical for the physical design process and manufacturability.

Each of these modalities provides a unique perspective of the circuit, and fusing them enables a comprehensive understanding of the design, facilitating advancing foundation AI techniques for circuits.

Multiple objectives. The ultimate target objectives in circuit design are *PPA* (i.e., power, performance, and area) and *functionality*. PPA metrics are crucial for optimizing the overall design, ensuring that the circuit meets the required performance standards while minimizing power

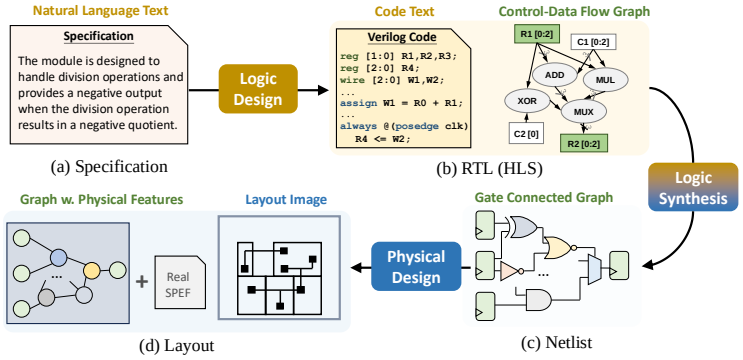


Fig. 4. VLSI design stages and corresponding modalities.

consumption and chip area. Functionality metric targets fulfilling the intended specifications, guaranteeing that the circuit behaves correctly under various conditions. Achieving both PPA optimization and functional correctness is essential for delivering robust and efficient hardware.

Parallel execution of hardware. Hardware circuits inherently operate with parallelism, clearly distinguishing them from the sequential execution of software code. In combinational logic, multiple logic operations are computed simultaneously, enabling high-speed parallel data processing. Meanwhile, sequential elements, such as registers and flip-flops, update synchronously at each clock cycle, ensuring efficient and coordinated circuit state transition. This fundamental parallelism plays a crucial role in defining circuit behavior, making it essential for accurately capturing circuit intrinsic properties in AI-driven design automation.

Circuit data availability. AI-driven EDA solutions depend on access to high-quality, diverse, and representative circuit data for both model development and evaluation. However, the scarcity of open circuit datasets remains a significant technical bottleneck. This challenge primarily arises from the semiconductor industry's reluctance to share proprietary circuit designs, which are considered valuable commercial IP. The absence of publicly available datasets hampers AI-driven EDA advancements, as collecting labeled data is both time-consuming and resource-intensive. Moreover, the limited diversity of open-source circuit designs restricts model generalization and performance.

As circuit foundation models gain traction in agile IC design, data availability becomes even more critical, particularly in the context of scaling laws for circuit foundation models, as observed in several existing works [97, 98]. These laws suggest that model performance improves with larger datasets, making the shortage of diverse and extensive circuit data a fundamental limitation in training highly capable models. Addressing this challenge is crucial for unlocking the full potential of AI-driven EDA solutions.

Circuit reusability. Reusability is a key factor in practical circuit development, as companies often rely on pre-designed IP blocks rather than building circuits from scratch. This inherent reusability presents an opportunity for circuit foundation models to exploit semantic similarities across designs, enhancing performance on downstream tasks. By leveraging patterns and shared features within circuit datasets, these models can improve efficiency and adaptability in various EDA applications.

4 FOUNDATION MODEL AS A CIRCUIT ENCODER

In this section, we will cover all encoder-based circuit foundation models. The circuit encoder paradigm consists of two major stages: (1) It first *pre-trains* AI models to encode circuits into generalized embedding vectors that capture rich intrinsic properties of circuits. These embeddings provide a flexible representation that can further be *fine-tuned* with task-specific supervision. (2) The *fine-tuning* process enables the embeddings to support various *predictive* downstream tasks, such as early-stage design quality prediction and functional reasoning, thus supporting design space exploration. The goal is to predict specific outcomes based on given circuit data. Figure 5 summarizes our covered encoder works based on their proposed pre-training techniques.

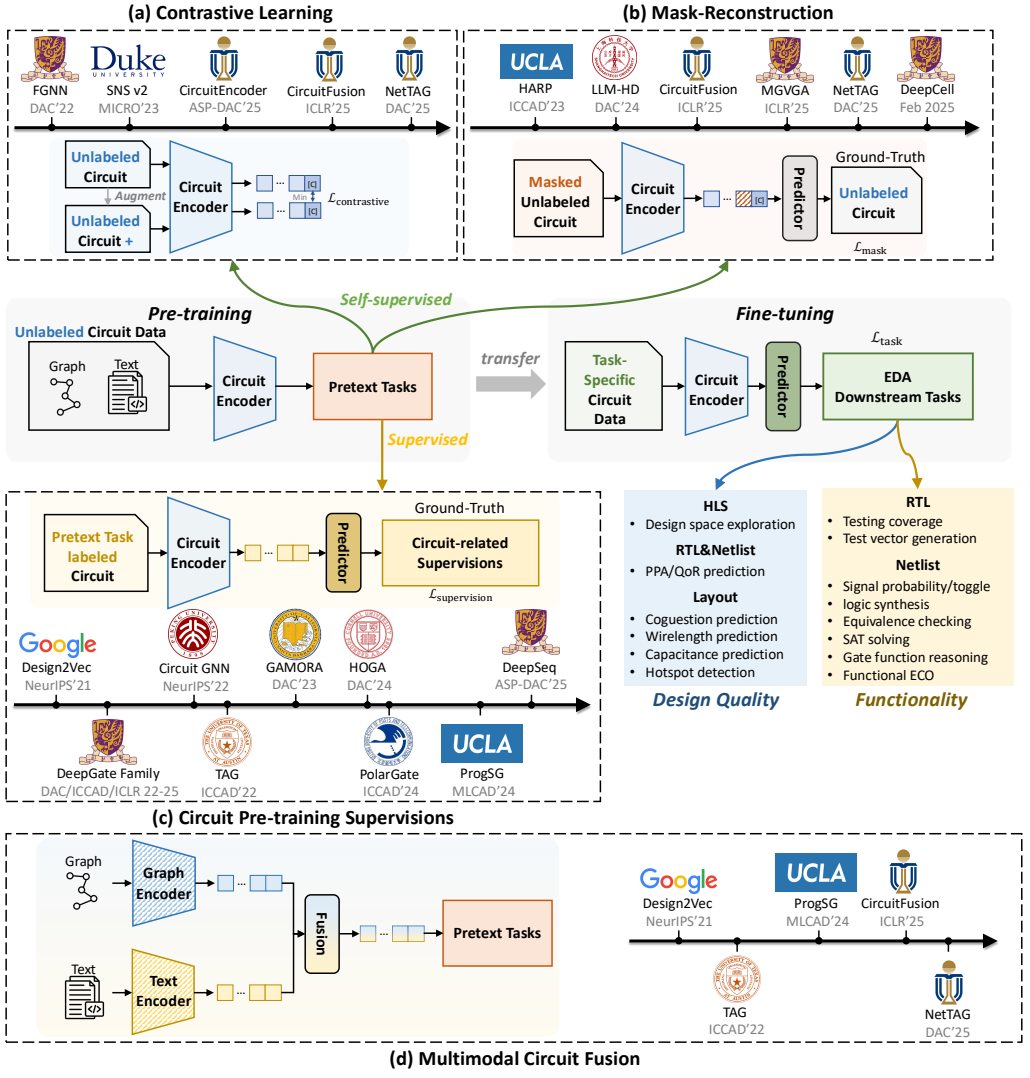


Fig. 5. Summary of pre-training techniques used in circuit encoders, covered in Section 4. Representative pre-training techniques include (a) self-supervised contrastive learning, (b) self-supervised mask-reconstruction, (c) circuit-related supervisions, and (d) multimodal circuit fusion.

Existing works mainly focus on exploring the pre-training techniques that capture the structural, semantic, and physical aspects of circuits. As summarized in Figure 5, we categorize the existing circuit encoder pre-training techniques into four types, each focusing on different aspects of learning circuit representations:

- **Self-supervised contrastive learning** in Figure 5 (a). This technique minimizes the distance between embeddings of similar circuits while maximizing the distance between embeddings of dissimilar circuits based on the circuits' functionality. This process pre-trains the model to differentiate between functionally equivalent and non-equivalent circuits. In this way, the pre-trained model learns meaningful representations that reflect the intrinsic functional properties of the circuit designs.
- **Self-supervised mask-reconstruction** in Figure 5 (b). In this technique, parts of the circuit representation are masked, and the model is pre-trained to reconstruct the masked missing parts. This process pre-trains the model to learn robust, complete representations of circuits, capturing both the structural and functional aspects. The pre-training task typically involves masking graph nodes or textual tokens of a circuit and using the remaining circuit information to predict the missing parts.
- **Supervised circuit pre-training tasks** in Figure 5 (c). In addition to self-supervised pre-training techniques, certain approaches incorporate task-related supervision to pre-train circuit encoders. Unlike direct target-task supervision in supervised methods, these pre-training tasks provide generalizable guidance to help the model learn circuit properties from labeled data. For example, predicting the truth-table distance between circuit pairs pre-trains the model to capture functional properties, which can then be leveraged for functional tasks such as SAT solving and logic synthesis.
- **Multimodal circuit fusion** in Figure 5 (d). This technique integrates multiple modalities of circuit data, such as textual, structural, and physical information, to create richer, more comprehensive representations. The model is pre-trained to fuse these different modalities, enabling it to capture a broader range of circuit characteristics. In this way, the model supports complex tasks that require information from different modalities.

In the following subsections, we summarize existing circuit encoders based on their target circuit design stages, including HLS stage (Section 4.1), RTL stage (Section 4.2), netlist stage (Section 4.3) and layout stage (Section 4.4). A detailed comparison and summary of these circuit encoders are provided in Table 2 and Table 3, respectively. Section 4.5 will cover the emerging and more advanced circuit encoder techniques. For each stage, we first summarize the employed circuit dataset, including detailed statistics and data collection process, then detail the proposed encoding techniques, including circuit preprocessing, ML model architecture, and pre-training techniques, and finally discuss the supported downstream tasks with evaluation metrics.

4.1 Circuit Encoder for HLS

In the context of HLS, the circuit encoder plays a pivotal role in representing and optimizing the design space for HLS circuits. HLS involves the transformation of high-level programming languages (e.g., C/C++) into hardware description languages (e.g., Verilog), with the goal of improving the design, performance, and power efficiency of hardware systems. Efficient exploration of this design space is critical, and HLS encoders are explored to learn meaningful representations of the circuit designs, enabling better optimization and decision-making. As shown in Figure 6 (a), two notable methods in this domain are HARP [99] and ProgSG [100], both pre-train HLS encoders with self-supervised learning, improving the exploration of the HLS design space.

4.1.1 Dataset for HLS circuits.

Target Stage	Method	Modality		Pre-Training		Downstream Task	
		Graph	Text	Self-Supervised	Supervised	Design Quality	Functionality
HLS	HARP [99]	✓		✓		✓	
	ProgSG [100]	✓	✓	✓		✓	
RTL	Design2Vec [101]	✓	✓		✓		✓
	SNS v2 [25]	✓				✓	
	CircuitEncoder [102]	✓		✓		✓	
	CircuitFusion [97]	✓	✓	✓		✓	
Netlist	DeepGate [103]	✓			✓		✓
	DeepGate2 [104]	✓			✓		✓
	DeepGate3/4 [98, 105]	✓			✓		✓
	GAMORA [54]	✓			✓		✓
	HOGA [106]	✓			✓	✓	✓
	PolarGate [107]	✓			✓		✓
	DeepSeq [108, 109]	✓			✓	✓	
	FGNN [110, 111]	✓		✓			✓
	CircuitEncoder [102]	✓		✓			✓
	MGVGA [112]	✓	✓	✓		✓	✓
	NetTAG [113]	✓	✓	✓		✓	✓
	DeepCell [114]	✓		✓			✓
Layout	Circuit GNN [115]	✓			✓	✓	
	TAG [116]	✓	✓			✓	
	LLM-HD [117]		✓	✓		✓	

Table 2. Comparison of modality, pre-training techniques, and supported downstream tasks for existing encoder-based circuit foundation models, as covered in Section 4.

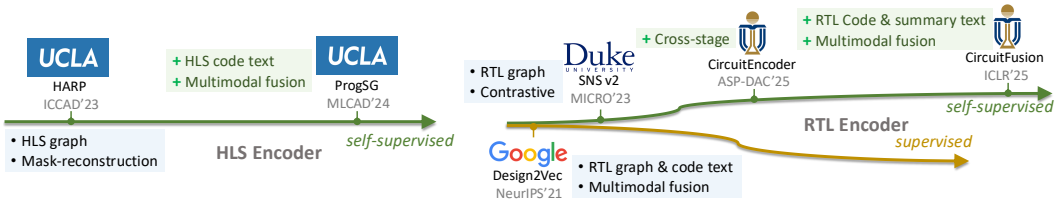
The HLS dataset [118] used in these works consists of 42 unique kernels, each with multiple optimization pragmas generated by the AMD/Xilinx HLS tool, resulting in over 10,000 design configurations. The HLS designs serve both text and graph modalities. In the text modality, the data consists of C/C++ code, averaging 1,286 tokens per program. In the graph modality, the programs are converted into the control-data flow graphs (CDFG), with an average of 354 nodes and 1,246 edges.

4.1.2 Encoding techniques for HLS circuits.

Both HARP [99] and ProgSG [100] employ self-supervised learning techniques to pre-train HLS encoders. HARP [99] focuses on encoding the graph format of HLS CDFG, while ProgSG [100] extends this by adding textual input for richer multimodal circuit representation learning. We detail the HLS encoding techniques below.

Self-supervised HLS graph encoder with masked pragma reconstruction. HARP [99] focuses on HLS control-data flow hierarchical graphs for representing circuit designs. Specifically, HARP [99] utilizes a hierarchical graph representation of HLS designs, incorporating both high-level and low-level views, where the high-level view combines C/C++ code and LLVM intermediate representation (IR) to capture the program's structure and semantics, and the low-level view focuses on LLVM IR to capture detailed implementation details. This dual-level representation helps mitigate long-range dependencies within the program. The model employs a GNN to encode this hierarchical graph into circuit embeddings. During pre-training, it applies a self-supervised learning technique called masked pragma reconstruction, with paradigm demonstrated in Figure 5 (b). In this approach, certain pragmas (compiler directives) are masked, and the GNN model is trained to predict these masked pragmas based on the surrounding node embeddings in the graph. This enables the model to learn the specific effects of each pragma, enhancing its performance and improving its ability to transfer knowledge across tasks.

Self-supervised HLS encoder enhanced via HLS graph-text mulimodal fusion. ProgSG [100] builds upon HARP [99] by integrating multimodal learning to improve HLS encoding. It combines two modalities: CDFG hierarchical graph used in HARP [99] and HLS C/C++ source code text,



(a) Timeline for HLS encoders (b) Timeline for RTL encoders

Fig. 6. Timeline for HLS (Section 4.1) and RTL (Section 4.2) encoders.

allowing the model to capture both structural and semantic aspects of the design. ProgSG [100] uses a GNN for graph encoding and an LLM for text encoding. It introduces a node-token message passing mechanism for multimodal fusion, where information is exchanged through block nodes and tokens from the high-level view before being propagated to normal nodes and tokens via GNN and transformer layers. To address the scarcity of labeled designs, ProgSG [100] employs a self-supervised pre-training technique based on compiler-generated data flow analysis tasks. This static analysis task predicts the relationship between two nodes in a CDFG, such as reachability and data dependencies, enabling the model to learn how data moves through the program. This pre-training improves the model's ability to generalize, boosting its performance in downstream tasks such as design space exploration and design optimization.

4.1.3 *Downstream tasks for HLS encoders.*

The two methods, HARP [99] and ProgSG [100], support downstream tasks that predict various HLS design quality metrics, including latency (in cycle counts), block RAM utilization, digital signal processor utilization, flip-flop utilization, and lookup-table utilization. These metrics are critical for evaluating the performance and efficiency of HLS designs. The models are assessed using the regression metric root mean square error (RMSE), which measures the accuracy of the design performance predictions.

In addition to performance prediction, these HLS encoders are further used for design space exploration, a task aimed at finding the optimal design for a given kernel. This process involves exploring various design configurations to identify the best-performing design in terms of resource utilization and latency.

4.2 Circuit Encoder for RTL Stage

In the RTL stage of VLSI design, the RTL encoder can capture both the semantics and structure of RTL circuits. As illustrated in Figure 6 (b), which shows the timeline of existing RTL encoders, four notable methods have emerged in this domain: Design2Vec [101] utilizes supervised pre-training tasks for functional verification tasks. In contrast, SNS v2 [25], CircuitEncoder [102], and CircuitFusion [97] employ self-supervised learning techniques for design quality prediction tasks.

4.2.1 *Dataset for RTL circuits.*

The RTL designs are used in both text and graph modalities: the text modality directly adopts the HDL code (e.g., Verilog), while the graph modality converts the RTL code into a CDFG based on the abstract syntax tree. For functional verification tasks, Design2Vec [101] employs three designs, including two RISC-V CPUs and one TPU. For each design, the authors generated random tests and sampled each test parameter uniformly. They used a testbench to randomly sample input test stimuli and a Verilog RTL simulator to obtain ground-truth labels of whether a cover point was covered by that test, resulting 4118 cover points in total. As for design quality evaluation tasks, SNS v2 [25] and

CircuitFusion [97] collect various types of RTL designs from various open-sourced benchmarks, including ITC'99 [119], OpenCores [120], Chipyard [121], VexRiscv [122], XiangShan [123], and other open-sourced designs. The RTL designs are synthesized using logic synthesis tools such as Synopsys Design Compiler, and the design quality metrics (i.e., PPA values) are obtained from the post-synthesis netlists. In the latest work CircuitFusion [97], the dataset scale includes up to 500K nodes for the circuit graph and up to 20M tokens for Verilog code text.

4.2.2 Encoding techniques for RTL.

As shown in Figure 6 (b), in the supervised encoding branch, Design2Vec [101] pioneers RTL encoding by learning functional semantics through pre-training supervisions for verification tasks. In the self-supervised learning branch, SNS v2 [25] proposes to leverage functional contrastive learning on RTL graphs to capture RTL circuit representations, while CircuitEncoder [102] enhances this by introducing cross-stage alignment with netlist stage, incorporating implementation details from netlists. CircuitFusion [97] further improves by integrating code text and functional summaries with the RTL graph for multimodal fusion, and adds additional self-supervised techniques to learn RTL circuits at multiple modalities and granularities. We detail the key techniques for RTL encoders below.

Supervised RTL semantic encoder with functional supervisions. Design2Vec [101] learns semantic representations of RTL circuits for functional verification. The input to Design2Vec [101] includes the hardware design represented as a CDFG derived from the RTL Verilog code, along with the corresponding source code text. The CDFG captures both the control and data flow aspects of the design, providing a comprehensive view of the hardware's functionality. Design2Vec [101] employs a GNN to process the RTL CDFG, with each node augmented by RTL code text embeddings obtained from an LSTM for multimodal fusion. To capture the sequence dependency of circuit functionality, an additional LSTM is used to generate final node embeddings. During pre-training, Design2Vec [101] uses a supervised learning pre-training task that predicts the coverage of specific points in the design when simulated on test inputs. This task requires the model to integrate both the structural and functional aspects of the hardware design, effectively learning the interactions between control and data flow.

Self-supervised RTL encoder with contrastive learning. Although the supervised pre-training task is designed to learn the circuit functionality, it cannot be generalized to other function-unrelated tasks. The other three RTL encoders (i.e., SNS v2 [25], CircuitEncoder [102], and CircuitFusion [97]) employ self-supervised learning techniques that learn a generalized circuit embedding. The pioneering work SNS v2 [25] first introduces self-supervised contrastive learning to learn generalized circuit embeddings. The input to the SNS v2 [25] model is the graph format of HDL code based on the abstract syntax tree. It proposes a hierarchical graph format for RTL designs, where the low-level graph consists of subgraphs sampled from registers, and the high-level graph represents register dependency. The model uses a two-level hierarchical GNN architecture. The low level processes small subgraphs, capturing local structural and functional features, while the high level aggregates these embeddings to predict quality metrics such as power, area, and timing for the entire design. For pre-training, SNS v2 [25] employs a contrastive learning approach to pre-train the subgraph GNN on unlabeled hardware designs. The model learns to create functionally equivalent circuit representations, where similar embeddings are assigned to functionally equivalent circuits, despite differences in their representation. This self-supervised learning task enables the model to understand circuit equivalence. After pre-training, the model is fine-tuned using labeled datasets and adapted to new domains, allowing it to predict design quality metrics for various RTL circuits.

Self-supervised RTL encoder enhanced with cross-stage alignment. Following SNS v2 [25], CircuitEncoder [102] also converts circuit RTL code into a graph-based representation using the abstract syntax tree. The model processes the graph using a graph transformer, which allows it

to learn from the structural relationships of RTL designs. For pre-training, CircuitEncoder [102] employs graph contrastive learning on RTL designs, similar to SNS v2 [25]. In addition, CircuitEncoder [102] introduces multi-stage contrastive learning, which involves learning embeddings both within the same design stage (intra-stage) and across different stages (inter-stage) between RTL and netlist designs. This technique helps align the embeddings from different design stages into a shared latent space, improving the model's ability to transfer learning between different stages and enhancing its generalization across the hardware design process.

Self-supervised RTL encoder enhanced with multimodal fusion. Another recent work CircuitFusion [97] proposes self-supervised learning and advances the RTL encoder by fusing multiple modalities of RTL designs to enhance chip design workflows. Specifically, it processes three input modalities: HDL code, representing circuit functionality in textual form (e.g., Verilog); graph format, capturing the circuit's structure through an abstract syntax tree; and functionality summary, a high-level textual abstraction of the design's function. The model uses unimodal encoders for each modality, including graph, code, and summary encoders, followed by a multimodal fusion encoder with a cross-attention mechanism to combine the outputs into a unified latent space. During pre-training, CircuitFusion [97] utilizes several self-supervised tasks: (1) Intra-modal learning, including contrastive learning and masked graph modeling to capture the internal structure of each modality, (2) Cross-modal alignment, where contrastive learning aligns the different modalities in a shared space, (3) Multimodal fusion, which involves tasks like masked summary modeling and mixup-embedding matching to combine structural and semantic information from all modalities, and (4) Implementation-aware alignment, which aligns RTL and netlist representations to ensure the design's functionality maps accurately to its physical implementation.

4.2.3 Downstream tasks for RTL encoders.

The RTL stage is crucial for implementing the functionality of the specification and serves as the foundation for design quality optimization, such as PPA. The primary downstream tasks for RTL encoders focus on functional verification and early-stage PPA prediction. For **functional verification**, the Design2Vec [101] model uses the semantic representations learned through pre-training tasks to predict whether a given test covers specific portions of the design and to generate test vectors. The model is evaluated based on its ability to predict coverage and detect bugs in hardware designs. By improving test generation and bug detection efficiency, the model enhances the overall verification process. For **design quality prediction**, SNS v2 [25], CircuitEncoder [102], and CircuitFusion [97] all focus on predicting key synthesis results, including area, power consumption, and timing for hardware designs. These models are evaluated using performance metrics such as the correlation coefficient (R) and Mean Absolute Percentage Error (MAPE), providing insights into the accuracy of design quality predictions and contributing to early-stage optimization for PPA.

4.3 Circuit Encoder for Netlist Stage

The netlist stage is one of the most actively explored stages in circuit encoders, with circuit encoding playing a critical role in extracting meaningful representations from the structural and functional properties of logic circuits. In recent years, several methods have been developed that apply graph learning techniques (e.g., GNNs and Graph Transformers), to improve netlist analysis. As shown in Figure 7 (a), the timeline for netlist encoders includes both supervised methods, such as the DeepGate Family [98, 103–105], HOGA [106], PolarGate [107], and DeepSeq [108, 109], as well as self-supervised methods like FGNN [110, 111], CircuitEncoder [102], NetTAG [113], and DeepCell [114]. These encoders have evolved from encoding simple AND-Inverter Graphs (AIGs) of netlists to more complex post-synthesis netlists involving various types of gates. For downstream tasks, these netlist encoders support a wide range of applications. These include

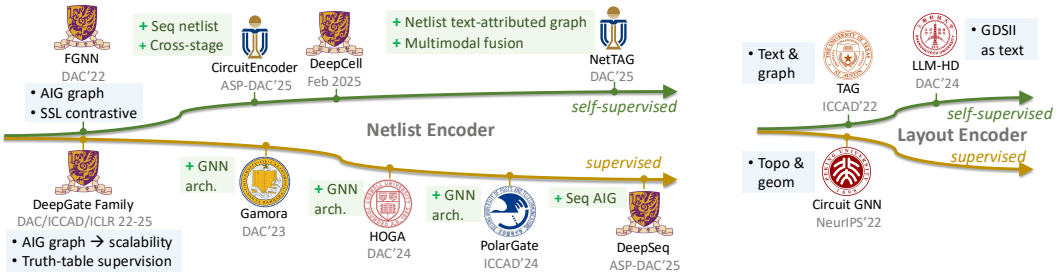


Fig. 7. Timeline for netlist (Section 4.3) and layout (Section 4.4) encoders.

functional reasoning and verification tasks, such as arithmetic block identification, SAT solving, and logic synthesis, as well as netlist-stage design quality evaluation tasks like timing, power, and area estimation.

4.3.1 Dataset for circuit netlists.

Most of the netlist encoders [98, 103–111] target the And-Inverter Graph (AIG) format of the netlist, which is an intermediate representation commonly used in logic synthesis and verification. Recently, works [113, 114] have expanded their scope beyond basic AIG gates to handle more complex post-synthesis netlists, which include various standard cells. For AIG datasets, the encoders gather data from various benchmarks like OpenABC-D [124], ITC'99 [119], IWLS [125], OpenCores [120], EPFL [126], GAMORA [54], arithmetic modules [110], Chipyard [121], and LGSynth-93 [127]. RTL designs from these benchmarks are typically converted into AIG formats using the ABC open-source logic synthesis tool. These encoders primarily focus on combinational logic within AIGs, with DeepSeq [109] also considering sequential registers in its encoding process. As for the post-synthesis netlist datasets, they are obtained from RTL benchmarks like ITC'99 [119], IWLS [125], OpenCores [120], EPFL [126], Chipyard [121] and VexRiscv [122]. Logic synthesis is conducted using technology libraries to generate the post-synthesis netlists. NetTAG [113] processes both combinational and sequential netlist gates, while DeepCell [114] focuses on the combinational aspects. This expansion enables the models to handle a wider range of netlist formats and to predict design quality more accurately at post-synthesis stages.

4.3.2 Encoding techniques for netlist.

As shown in Figure 7 (a), we categorize the encoding methods into supervised pre-training tasks and self-supervised learning methods. In the supervised encoding branch, DeepGate family [98, 103–105] pioneers AIG encoding for netlists, learning functional semantics for logic synthesis and verification tasks. They have improved scalability with advanced supervision, better graph learning models, and optimized memory consumption. Other methods, like HOGA [106] and PolarGate [107], refine AIG encoding with customized GNN architectures and message-passing mechanisms to capture both structural and functional properties. DeepSeq [108, 109] extends the DeepGate Family to handle sequential circuits, improving the model's ability to process more complex circuit behaviors. In the self-supervised learning branch, FGNN [110, 111] first introduces functional contrastive learning to solve the arithmetic block identification problem. CircuitEncoder [102] enhances this with cross-stage alignment, incorporating RTL-stage information to improve netlist encoding. NetTAG [113] and DeepCell [114] push the boundaries of AIG encoding by advancing it to handle more complex post-synthesis netlists, allowing for the processing of designs with various

Target Stage	Method	Technique Pre-train objective	Downstream Task
HLS	HARP [99]	Masked pragma reconstruction	HLS design space exploration
	ProgSG [100]	Data flow analysis tasks for graph and node	HLS design space exploration
	Design2Vec [101]	Testing cover point prediction	Verification coverage prediction and test generation
RTL	SNS v2 [25]	Functional contrastive learning	Post-synthesis PPA prediction
	CircuitEncoder [102]	Intra-stage functional contrastive learning Cross-stage functional contrastive alignment	Post-synthesis PPA prediction
	CircuitFusion [97]	Masked gate reconstruction Functional contrastive for graph/ summary Modality fusion Cross-design-stage alignment	Post-synthesis PPA prediction
Netlist	DeepGate [103]	Signal probability prediction	Signal probability prediction on large AIGs
	DeepGate2 [104]	Truth-table supervisions on node	Logic synthesis and SAT solving
	DeepGate3/4 [98, 105]	Truth-table supervisions on node and graph	SAT solving
	GAMORA [54]	Task-specific supervisions	Logic functional reasoning
	HOGA [106]	Task-specific supervisions	Logic synthesis QoR prediction, functional reasoning
	PolarGate [107]	Truth-table supervisions	Signal probability and truth-table distance prediction
	DeepSeq [108, 109]	Truth-table supervisions on node	Toggle rate prediction for power analysis
	FGNN [110, 111]	Functional contrastive learning	Gate function reasoning
	CircuitEncoder [102]	Intra-stage functional contrastive learning Cross-stage functional contrastive alignment	Register function reasoning
	MGVGA [112]	Masked gate reconstruction	QoR prediction, logic equivalence identification
	NetTAG [113]	Logic expression contrastive Masked gate reconstruction Netlist graph contrastive learning Netlist graph size prediction	Post-layout PPA prediction Gate/Register function prediction
	DeepCell [114]	Masked circuit modeling	Functional ECO
Layout	Circuit GNN [115]	Task-specific supervisions	Congestion and wirelength prediction
	TAG [116]	Layout instance distance prediction	Wirelength, and net parasitic capacitance prediction
	LLM-HD [117]	Masked language modeling	Hotspot detection

Table 3. Summary of the pre-training techniques and supported downstream tasks of circuit encoders, covered in Section 4.

standard cells and more intricate gate structures, thus improving the prediction and optimization of hardware designs in post-synthesis stages.

Supervised AIG encoder with functional supervision. The DeepGate family [98, 103–105] is one of the pioneers in netlist encoders. They handle circuit AIGs using customized graph learning models, which are pre-trained using supervised pre-training tasks. These works primarily focus on functional-related tasks, such as training on pairwise truth table differences between sampled logic gates. The DeepGate family continuously improves model performance and scalability, with DeepGate3 [98] introducing a graph transformer to capture global circuit relationships, and DeepGate4 [105] optimizing the model by eliminating redundant computations.

Specifically, DeepGate [103] employs a GNN architecture specifically tailored for AIG graphs, incorporating an attention mechanism and recurrent layers to aggregate information across the graph. Each node's embedding is computed based on its gate type and its relationships with neighboring nodes. The recurrent GNN is designed to capture the functional behavior of the circuit by using both forward and reversed propagation layers, simulating the logic behavior. Signal probability (the probability of a node being in logic '1') is used as the supervision task, with signal probabilities derived from random logic simulations. These simulations are run on the circuits to obtain accurate probability values, allowing the model to learn functional behavior more effectively.

DeepGate2 [104] enhances the functionality-awareness encoding by introducing the Hamming distance between the truth tables of logic gates as supervision. The model uses a one-round GNN architecture, which processes both functional and structural embeddings for each gate. Unlike the multi-round GNN used in the original DeepGate, this one-round architecture efficiently propagates embeddings in a single pass. The functional embeddings represent the logic behavior of gates, incorporating the pairwise truth table difference as a supervisory signal, while structural

embeddings capture the topology of the circuit. A self-attention mechanism is used to aggregate information across different gates, enabling the model to focus more on controlling fan-in gates. During pre-training, a functionality-aware loss is proposed, which aligns gate embeddings with their functional equivalence. This loss minimizes the distance between embeddings corresponding to gates that perform similar logical operations, thereby improving the model's ability to recognize functionally equivalent gates.

DeepGate3 [98] improves upon DeepGate2 [104] by enhancing both performance and scalability with a graph transformer model. It uses DeepGate2 [104] as the AIG node tokenizer and refines the node embeddings with a graph transformer to capture long-range dependencies within the graph. For generating graph-level embeddings, another graph transformer is used for pooling. During pre-training, in addition to the gate-level supervisions used in DeepGate2, graph-level tasks are introduced. These tasks involve using fan-in cones to segment circuits into smaller subgraphs and predict intrinsic features such as the size and depth of these subgraphs, further improving the model's ability to understand circuit structures at a broader level.

DeepGate4 [105] further improves the scalability and efficiency challenges of large-scale circuit AIG representation learning by integrating a GAT-based sparse transformer. By leveraging graph sparsity, the model reduces the time and memory complexity of the transformer, making it suitable for processing large circuits. The architecture also incorporates structural encodings for gates, such as level and out-degree, to enhance the learning of circuit properties. The circuit graph is partitioned into smaller cones based on logic levels, which are then processed by the sparse transformer. This approach significantly improves both accuracy and computational efficiency, particularly for large-scale circuit designs, outperforming previous methods in terms of scalability and overall performance.

Supervised AIG encoder enhanced with GNN architecture. In addition to DeepGate family, other works (i.e., GAMORA [54], HOGA [106] and PolarGate [107]) explore to customize the GNN architecture and message-passing mechanism to enhance the scalability and performance of AIG encoding, combining with supervised pre-training tasks. In GAMORA [54], netlist AIGs are transformed into a graph representation and processed using a GNN. During pre-training, the GNN is trained with multiple functionally driven tasks, which jointly reason about Boolean function aggregation and structural topology. This enables efficient symbolic reasoning for large-scale Boolean networks. Specifically, GAMORA [54]'s model is designed to recognize fundamental functional components within circuits, including identifying adder root and leaf nodes and detecting XOR and MAJ functions. The multi-task learning framework enhances the model's ability to generalize across various functional tasks, leveraging shared representations to improve both accuracy and scalability in processing large AIG-based netlists.

In HOGA [106], hop-wise features are precomputed for each design to capture interactions over multiple hops before training. This step is done independently of the graph structure, enabling scalability for distributed training. The AIG format of circuits is processed using a customized GNN with a hop-wise aggregation scheme, which precomputes features based on multiple hops. It also employs gated self-attention to adaptively learn high-order circuit structures. This approach avoids recursive aggregation, which can be computationally expensive for large circuits. The model is then trained using task-specific labels, allowing it to be adapted for downstream tasks.

In PolarGate [107], each node in the netlist AIGs represents two logical states: low level (0) and high level (1), which are fundamental for Boolean logic tasks. The model employs a GNN with a novel functionality-aware message passing mechanism that aggregates information from neighboring nodes while distinguishing between AND and NOT gates through specialized operators. To achieve this, PolarGate [107] introduces an ambipolar embedding space, where each node is mapped to both a positive and a negative embedding to represent the two logical states. It also uses differentiable

logical operators, such as OPAND and OPNOT, that are designed to be differentiable and compatible with embedding propagation in the AIG structure. Additionally, the message passing strategy is modified to adhere to Boolean logical behavior, ensuring more accurate functional representation of the circuit. These innovations enable PolarGate [107] to effectively capture the logical operations of circuits and improve the model's ability to process and learn from netlist-based designs.

Supervised AIG encoder enhanced for sequential circuits. Beyond focusing on the combinational logics of AIGs, DeepSeq [109] explores capturing the sequential behavior of AIGs. DeepSeq [109] further advances this by using a directed acyclic GNN, which is optimized for sequential netlists. It incorporates a customized propagation scheme that avoids recursive propagation and handles cyclic sequential netlists in a single forward pass. The architecture separates learning into three distinct embedding spaces: structure embedding for circuit connectivity, function embedding for logic computations, and sequential embedding for capturing the temporal behavior between consecutive clock cycles. During pre-training, DeepSeq [109] uses multiple functional pre-training supervisions, including transition probability prediction to model sequential behavior, logic probability prediction to capture logic functionality, and pairwise truth-table difference to identify functional similarities among logic gates. These techniques enable DeepSeq [109] to effectively learn both the functional and sequential aspects of sequential AIG circuits.

Self-supervised AIG encoder with contrastive learning. In addition to customized supervised pre-training tasks based on circuit properties, another key approach for netlist encoders is leveraging self-supervised learning techniques to learn from unlabeled circuit data and capture the intrinsic information of the circuit. FGNN [110, 111] is a pioneer in adopting self-supervised contrastive learning for AIG netlist encoding. It uses a customized GNN to encode AIGs into embeddings, integrating a contrastive learning framework to enhance circuit functionality learning. The GNN architecture incorporates two types of learnable message aggregators: an ANG aggregator for AND gates and an INV aggregator for inverters. Asynchronous message passing is employed to efficiently propagate information through the graph while preserving functionality semantics. During pre-training, the model uses a contrastive learning scheme to learn circuit embeddings that reflect the Boolean functionality of the circuits. This scheme ensures that the embeddings of functionally equivalent circuits are close in the embedding space. Additionally, a new loss function is introduced to effectively capture the relative functional distance between circuits, taking into account input order invariance and circuits with different input widths, further improving the model's ability to represent the circuits' functionality.

Self-supervised netlist encoders with cross-stage alignment. Recent advancements in self-supervised learning for netlist encoders have introduced cross-design-stage alignment to enhance model awareness of different abstraction levels in circuit design. CircuitEncoder [102] and MGVGA [112] both propose novel alignment strategies to bridge the high-level abstract semantics of RTL with the low-level implementation details of netlists, enabling more robust circuit representation learning. CircuitEncoder [102] represents both RTL and netlist circuits as graph-based structures and processes them using a GNN. To learn circuit intrinsics, the model employs graph contrastive learning within each design stage, differentiating functionally similar and dissimilar circuit graphs. Additionally, it introduces intra-stage contrastive learning between RTL and netlist stages, effectively aligning representations across design stages. This cross-stage awareness significantly improves the encoder's adaptability for downstream tasks following fine-tuning. Similarly, MGVGA [112] proposes the concept of RTL-netlist alignment by integrating LLM-based processing for RTL descriptions and GNN-based encoding for AIG netlists. During pre-training, it introduces masked gate modeling, a technique that masks gates in the latent space while preserving logical equivalence, ensuring functional consistency throughout the representation learning process. Furthermore, a cross-modal learning strategy is implemented, where Verilog-based functional

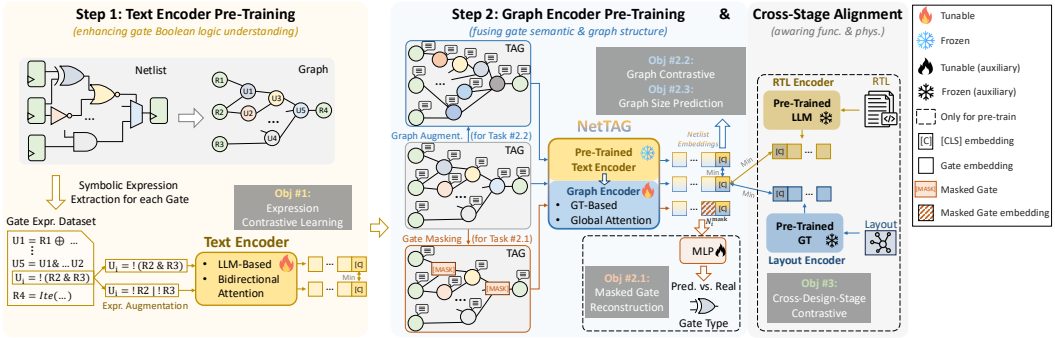


Fig. 8. Multimodal pre-training techniques used in NetTAG [113], including representative self-supervised learning methods such as contrastive learning, mask-reconstruction, and cross-design-stage alignment.

constraints guide AIG-based representation learning, reinforcing the structural-functionality alignment. These cross-stage alignment techniques enhance the capability of netlist encoders, improving their generalization across multiple circuit design stages and boosting performance in downstream EDA applications.

Self-supervised post-synthesis netlist encoder with multimodal fusion. While many existing netlist encoders focus on simpler AND-Inverter gates, they struggle with more complex post-synthesis netlists that involve various gates from standard liberty cells. To address this challenge, two recent works have advanced netlist encoding by incorporating multimodal fusion (i.e., NetTAG [113]) or AIG-netlist alignment (i.e., DeepCell [114]) to unprecedentedly handle the complexities of post-synthesis netlists.

As shown in Figure 8, in NetTAG [113], post-synthesis netlists are represented as text-attributed graphs, where each node corresponds to a gate and is associated with attributes that include both functional symbolic logic expressions and physical characteristics (such as area, power, and delay). The model employs a two-stage multimodal hybrid architecture: first, an LLM-based text encoder processes the textual attributes of the gates to generate semantic-rich embeddings. Then, a graph transformer refines these embeddings by capturing the global circuit structure through graph-based attention mechanisms. During pre-training, the model utilizes four key self-supervised objectives. Expression contrastive learning enhances the LLM's understanding of Boolean logic by contrasting symbolic expressions. Masked gate reconstruction is a graph-based task where certain gates are masked, and the model predicts the gate type, encouraging it to capture structural roles. Netlist graph contrastive learning aims to group similar netlists together while separating dissimilar ones, improving the model's ability to recognize functional equivalence in different netlist structures. Finally, cross-stage contrastive alignment aligns netlist embeddings with RTL and layout embeddings, combining functional and physical information to improve performance across various design stages. These self-supervised tasks enable NetTAG [113] to learn both the functional and structural aspects of post-synthesis netlists, significantly enhancing its ability to predict design qualities and optimize circuits across different stages of the design process.

Self-supervised post-synthesis netlist encoder with AIG-netlist alignment. DeepCell [114] proposes multiview representation learning to simultaneously capture structural and functional information from both post-synthesis netlists and AIGs. The model uses two separate encoders: the PM Encoder, which is a GNN designed to capture the features of standard cells from the post-synthesis netlists, integrating both structural and functional embeddings through specialized aggregators, and the AIG Encoder, which is a pre-trained AIG encoder based on DeepGate2 [104] that generates gate-level embeddings to provide additional structural information. During pre-training, DeepCell [114] employs a self-supervised mask circuit modeling task, where a subset of the cell

embeddings is masked, and the model reconstructs these embeddings using the information from the AIG encoder. This approach refines the post-synthesis netlist representations by integrating insights from both the local circuit view (i.e., netlist) and the global gate-level view (i.e., AIG), enhancing the overall quality of the circuit representation and improving downstream tasks such as design quality prediction and functional verification.

4.3.3 Downstream tasks for netlist encoders.

Netlist encoders support a wide range of downstream tasks, including functional reasoning and verification tasks, as well as netlist-stage design quality evaluation tasks such as timing, power, and area estimation. In **functional reasoning and verification**, key tasks ensure circuit functional correctness. Logic probability prediction estimates the likelihood that a gate outputs a logic '1', evaluated by Mean Absolute Error (MAE). Equivalence class identification groups functionally equivalent gates, with performance assessed by classification accuracy. SAT solving checks Boolean satisfiability, evaluated by solving time and satisfiability accuracy. Arithmetic function block identification identifies components like adders, assessed using classification metrics. Finally, functional ECO identifies mismatches post-synthesis, with evaluation based on error reduction and change cost. In **design quality prediction**, tasks focus on estimating key metrics like power, area, and delay. Logic synthesis QoR prediction uses MAPE to predict power, area, and delay after synthesis. Power evaluation estimates power based on toggle rate, evaluated by MAE and accuracy. Post-layout PPA prediction estimates power, performance, and area after layout, using MAPE to compare predicted versus actual results. These tasks enhance circuit optimization and validation.

4.4 Circuit Encoder for Layout Stage

In the layout stage of hardware design, circuit encoders process either the netlist or the GDSII format of circuit layouts. As shown in Figure 7 (b), the timeline for layout encoders includes both supervised methods such as Circuit GNN [115] which handles layout topology and geometry, and self-supervised methods like TAG [116], which employs text-graph multimodal encoding, and LLM-HD [117] which treats layout GDSII data as text. These methods focus on effectively capturing the physical and structural properties of layout designs to improve design quality prediction.

4.4.1 Dataset for layout circuits.

The datasets used in Circuit GNN [115] come from the ISPD2011 benchmark [128] for congestion prediction (12 designs) and the DAC2012 dataset [129] for net wirelength prediction (7 designs). These are preprocessed into a Circuit Graph that combines topological and geometrical information. TAG [116] uses 447 industrial AMS circuits in sub-10nm technology. The data is processed with StarRC extraction tools to obtain placement coordinates and create spatial and text embeddings by annotating the netlists with instance names and device types. For LLM-HD [117], the ICCAD 2012 [130] and ICCAD 2020 [131] benchmarks are used for layout hotspot detection, with GDSII layouts. The ICCAD 2012 dataset [130] focuses on metal layer hotspots, and ICCAD 2020 [131] on via-layer patterns. The data is processed directly from GDSII to preserve spatial and geometric features, using semantic and hierarchical encoding.

4.4.2 Encoding techniques for layout.

Figure 7 (b) illustrates the categories of layout encoders. In the supervised branch, Circuit GNN [115] customizes the GNN architecture to capture both the topology and geometry of the circuit layout. In the self-supervised branch, TAG [116] proposes text-graph multimodal learning, while LLM-HD [117] focuses on leveraging LLMs for textual layout encoding. These approaches enable the models to effectively learn and represent the topology, geometry, and physical property of circuit layouts for downstream tasks.

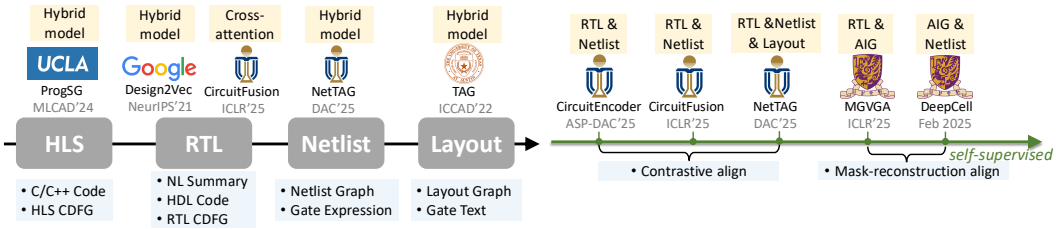
Supervised layout encoder with customized GNN architecture. In Circuit GNN [115], the input modalities consist of topological data (from the netlist) and geometrical data (from the layout). These modalities are represented in a circuit graph, where cells and nets are the vertices, and topo-edges and geom-edges connect them. The model is built upon a GNN, which processes the heterogeneous circuit graph. The GNN utilizes message-passing to propagate information across both topological and geometrical edges. Topological information is passed through topo-edges, while geometrical information is transmitted via geom-edges. These messages are then fused to update the representations of cells and nets. During pre-training, the integration of topological and geometrical information is achieved through the message-passing paradigm, with topo-geom message-passing ensuring both types of data contribute to the final learned representation. Task-specific supervisions are employed to train the model.

Self-supervised layout encoder with text-graph multimodal fusion. TAG [116] framework employs three primary modalities: (1) Text embedding, where the instance names and device/sub-circuit types from the circuit netlists are used as text inputs, processed using fastText to generate word embeddings, (2) Graph format, where the circuit is represented as a heterogeneous hierarchical graph encoding devices (nodes) and their connections (edges), including device types (e.g., NMOS, PMOS, capacitors) and hierarchical relationships between sub-circuits, and (3) Self-attention, where a multi-head self-attention layer is applied to the embeddings to capture global dependencies between instances within sub-circuits. The model architecture combines a GNN with self-attention to process both graph and text embeddings. GNN layers aggregate node information, while the self-attention mechanism ensures a global view of the circuit by considering the entire sub-circuit during training. During pre-training, the model is trained in an unsupervised manner with a focus on predicting the relative layout distance between instances within a sub-circuit. This distance prediction task is framed as a regression problem, where the embeddings are trained to predict the normalized relative distance between instances in manual layouts.

Self-supervised layout encoder with text semantic encoding. In LLM-HD [117], the input modalities of this layout encoder include GDSII layout data and its semantic encoding. The GDSII data is transformed into a sequential format to make it suitable for processing by a language model. The key components include polygon shapes and spatial relationships between them, encoded as sequential tokens. The model architecture employs a BERT-based transformer specifically designed for layout patterns, utilizing multi-head self-attention to capture relationships between layout features both locally and globally. The architecture consists of an embedding layer, followed by LLM-HD [117] layers, and concludes with a classification layer. During pre-training, the model uses masked language modeling, an unsupervised task where portions of the input data are randomly masked, and the model predicts the masked portions. This pre-training enables the model to learn representations of layout patterns, before fine-tuning for specific tasks such as hotspot detection.

4.4.3 Downstream tasks for layout encoders.

Circuit GNN [115] supports both congestion prediction and net wirelength prediction tasks. For congestion, it predicts routing congestion during both the logic synthesis and placement stages, evaluated using correlations and classification metrics like precision, recall, and F1-score. TAG [116] handles three layout-stage tasks: layout matching prediction (binary classification of layout constraints, evaluated by accuracy, TPR, FPR, PPV, and F1-score), wirelength estimation (HPWL evaluated with R^2, MAE, and sMAPE), and net parasitic capacitance prediction (evaluated using R^2 and MAE). LLM-HD [117] focuses on hotspot detection, a binary classification task identifying layout areas prone to manufacturing defects.



(a) Multimodal fusion (b) Cross-design-stage alignment

Fig. 9. Timeline for multimodal fused and cross-stage aligned encoders.

4.5 Summary of Trending Techniques for Advancing Circuit Encoders

4.5.1 Trend 1: Customized ML model architecture and pre-training tasks for circuits.

ML model architecture customized for circuits. To effectively capture the unique structural and functional properties of circuit data, various customized architectures have been developed, particularly in graph-based learning models. These architectures integrate specialized message-passing mechanisms to enhance the representation of circuit structures. For instance, GAMORA [54] and PolarGate [107] introduce customized GNN-based message passing tailored for AIGs, enabling the model to efficiently capture both Boolean functionality and structural connectivity.

Pre-training tasks customized for circuits. Pre-training tasks for circuit encoders can be divided into supervised and self-supervised methods, with each method specifically designed to capture the unique properties of circuit data. In supervised learning, for example, DeepGate2 [104] uses truth table supervision to train encoders by comparing the pairwise differences between truth tables of logic gates, thereby capturing the functional behavior of the circuit. This approach helps the encoder learn how different gates function in the context of their logic operations. On the other hand, for example, SNS v2 [25] introduces self-supervised learning for RTL encoders with functional contrastive learning. The model learns to cluster functionally similar circuits and separate dissimilar ones in the latent space. Another notable self-supervised pre-training task is masked circuit reconstruction, such as used in NetTAG [113], where specific gates in a circuit's netlist are masked, and the model learns to predict the missing gates based on the surrounding context. These pre-training tasks are vital for learning generalized representations that can be fine-tuned for various downstream tasks, such as design space exploration and functional verification.

4.5.2 Trend 2: circuit multimodal fusion.

Circuit design involves multiple modalities, including hardware description languages, graphical representations, and functional summaries, each capturing different aspects of the circuit. Recent works in circuit foundation models have focused on integrating these modalities through multi-modal fusion techniques, enabling models to leverage both structural and semantic information for more robust representation learning. Two primary approaches have emerged in this area: hybrid ML model architecture that combines different encoders for various modalities and cross-attention-based fusion with self-supervised learning. The timeline of multimodal fused encoders is demonstrated in Figure 9 (a).

Multimodal fusion by hybrid models. Hybrid models integrate distinct encoding architectures tailored to specific circuit modalities. For example, ProgSG [100] and Design2Vec [101] focus on HLS and RTL-stage circuits, respectively, where the source code contains rich semantic information. These models employ LLMs to encode textual descriptions while using GNNs to capture structural information from control-data flow graphs. This dual-modality approach ensures that both functional intent and circuit topology are preserved in the learned representations. At the netlist and layout stages, where the netlist code provides limited functional information, NetTAG [113]

and TAG [116] adopt a similar hybrid approach but with modifications suited for lower-level representations. NetTAG [113] extracts detailed symbolic logic expressions for each gate, encoding them using an LLM, while a GNN captures the circuit's global structural dependencies. TAG [116] follows a similar strategy, leveraging textual attributes alongside graph-based structural encodings to improve netlist representation.

Multimodal fusion by cross attention. In addition to hybrid models, cross-attention-based fusion has been proposed as an alternative strategy for multimodal integration. CircuitFusion [97], designed for RTL-stage encoding, processes three primary modalities—HDL code, functional summaries, and structural graphs—in parallel. Each modality is first encoded independently, after which an additional multimodal fusion encoder with cross-attention mechanisms aligns and integrates the learned representations. The cross-attention mechanism ensures that the fused representation retains complementary information from all modalities while mitigating redundancy. CircuitFusion [97] further enhances representation learning through self-supervised tasks such as masked summary modeling and embedding mixup, reinforcing the alignment between modalities.

4.5.3 Trend 3: cross-design-stage alignment.

Cross-design-stage alignment has become a promising direction in circuit foundation models, enabling representations learned at earlier design stages (e.g., RTL) to be aligned with their corresponding lower-level implementations (e.g., netlist, layout). This alignment enhances generalizability, allowing models to better capture the functional and physical characteristics of circuits throughout the design process. Two primary approaches have been explored for achieving cross-stage alignment: contrastive learning-based alignment and mask-reconstruction-based alignment. The timeline of cross-stage aligned encoders is demonstrated in Figure 9 (b).

Cross-stage alignment via contrastive learning. Contrastive learning-based alignment has been effectively used for bridging different design stages by learning stage-invariant circuit representations. CircuitEncoder [102] and CircuitFusion [97] focus on RTL-to-netlist alignment by integrating structural and functional representations through self-supervised contrastive learning. These models enforce similarity constraints between functionally equivalent circuits across design stages, ensuring that embeddings capture both high-level design intent and low-level implementation details. NetTAG [113] extends this approach beyond RTL and netlist, incorporating layout information to enable RTL-netlist-layout alignment. By leveraging cross-modal contrastive learning, NetTAG [113] aligns representations across all three design stages, facilitating more accurate early-stage predictions of post-layout circuit characteristics.

Cross-stage alignment via mask-reconstruction. Mask-reconstruction-based alignment, on the other hand, focuses on recovering masked portions of a circuit while maintaining logical and structural consistency across design stages. MGVGA [112] is designed for RTL-to-AIG alignment, where it employs Masked Gate Modeling to selectively mask gate-level details in AIG representations while preserving functional equivalence. This ensures that the learned embeddings retain both RTL-level semantics and AIG-level logic properties. DeepCell [114] also employs this technique for AIG-to-netlist alignment, incorporating a self-supervised masking strategy to reconstruct standard cell representations from their lower-level gate descriptions. This approach enhances the model's ability to understand structural variations while preserving functional equivalence across abstraction levels.

5 FOUNDATION MODEL AS A CIRCUIT DECODER

Another major paradigm of circuit foundation models is circuit decoders, which leverage LLMs for the automated generation of circuit-related content. These models facilitate the creation of RTL code (e.g., Verilog or VHDL), HLS code (e.g., SystemC or C++), design scripts (e.g., Tcl or Python), design descriptions, etc. As summarized in Figure 10, this section provides a comprehensive overview of

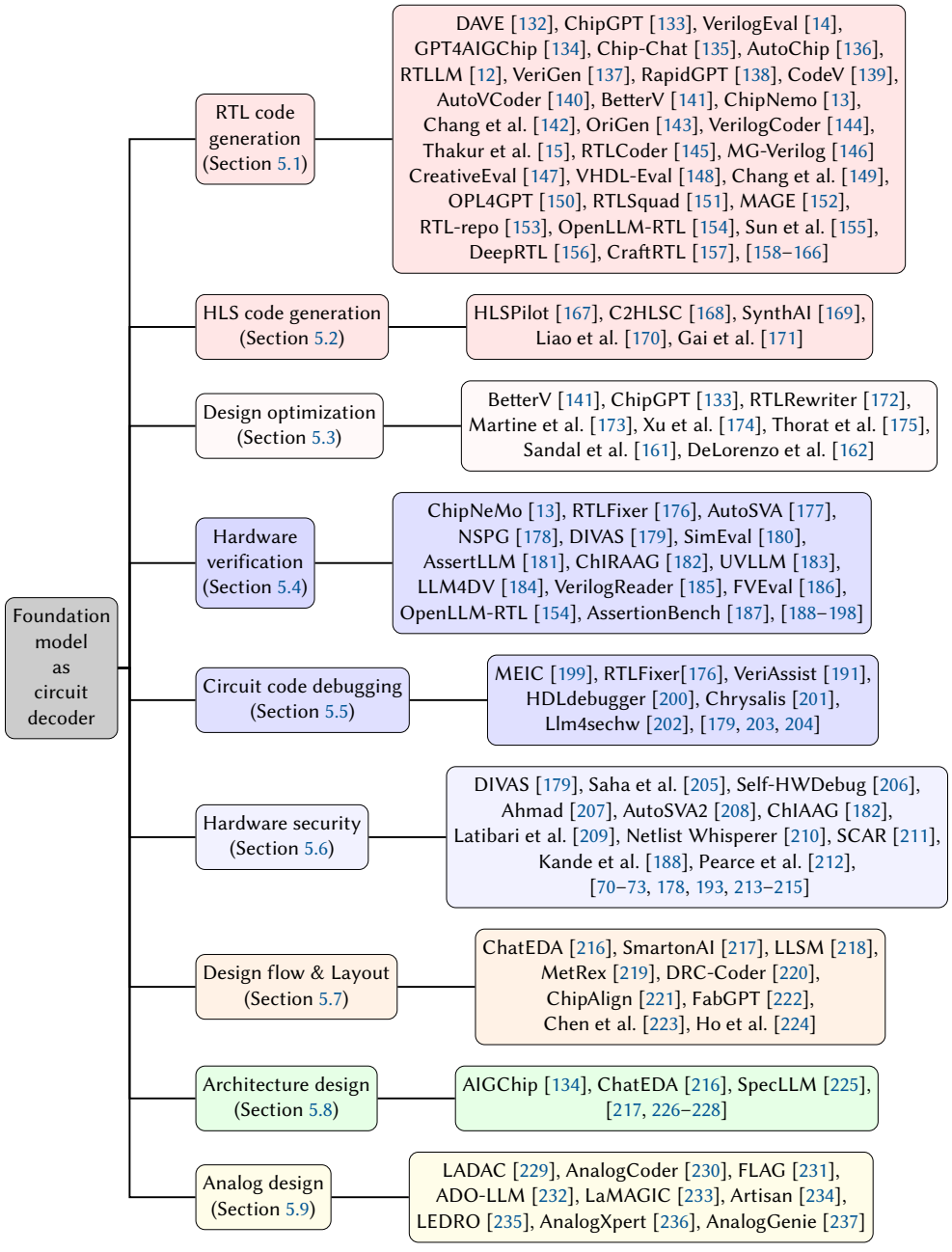


Fig. 10. Research tree of foundation models as circuit decoder, covered in Section 5.

LLM-assisted circuit design techniques, categorizing them into 7 main directions according to their applications in generative EDA tasks:

(1) **LLM-assisted hardware code generation and optimization.** This category explores the application of LLMs in generating hardware code across different abstraction levels, such as RTL and HLS code. We will discuss the use of LLMs for RTL code generation in Section 5.1 and HLS code generation in Section 5.2. Additionally, we will examine efforts in producing *optimized* hardware in Section 5.3. Some works in this *optimization* category overlap with the hardware code *generation*.

(2) **LLM-assisted hardware code verification and debugging.** Beyond code generation, LLMs are employed to verify the correctness of hardware code (HLS or RTL) and to fix potential bugs. Section 5.4 will cover the role of LLMs in hardware code *verification*, while Section 5.5 will focus on hardware *debugging* techniques.

(3) **LLM for hardware security.** Works on *hardware security* focus on security-oriented design, debugging, and verification. We will delve into these topics in Section 5.6, highlighting the unique challenges in security as distinct from general verification and debugging.

(4) **LLM for design flow automation and layouts.** Section 5.7 explores the application of LLMs in automating the *design flow* based on natural-language instructions, as well as enhancing circuit *layout* processing for improved manufacturability.

(5) **LLM for hardware architecture design.** Section 5.8 addresses the application of LLMs at a higher level of abstraction, focusing on design *architecture* and *specifications*. This includes applications for memory design and AI accelerators.

(6) **LLM for analog circuit design.** Beyond the scope of digital VLSI design, LLM-assisted *analog circuit* design is another important direction. In Section 5.9, we will explore how LLMs can benefit analog circuit design, highlighting the significance of this area in the broader context of hardware development.

5.1 LLM for RTL Code Generation

RTL design is a crucial step in the whole VLSI design process. This process defines the expected behavior of circuits with hardware description languages (HDLs) like Verilog and VHDL. However, RTL design remains a manual, time-consuming, tedious, and error-prone task. Recently, leveraging LLMs for RTL generation offers a promising automated solution. Specifically, LLM solutions can directly generate expected design RTL in HDL code, typically based on design descriptions in natural language as LLM input. Such *RTL code generation* is the most extensively explored application of LLM-assisted EDA techniques. The existing works contribute primarily in two ways: 1) new *benchmarks* evaluating LLM performance, covered in Section 5.1.1 and listed in Table 4; and 2) new *LLM solutions* on RTL code generation, covered in Section 5.1.2 and listed in Table 5 and Figure 12.

5.1.1 RTL code generation benchmarks.

As LLMs become popular for RTL design generation, *benchmarks* become crucial for assessing the accuracy, efficiency, and reliability of LLM-based solutions for circuits. We summarize all benchmarks on RTL generation in Table 4, among which RTLLM [12] and VerilogEval [14] are two *pioneering and most widely-adopted* benchmarks for evaluating RTL code generation. Figure 11 illustrates the evaluation process of the RTL code generation benchmarks. A typical benchmark [12, 14] will provide dozens of design cases, each corresponding to one small circuit design or component. For each case, the benchmark will provide three types of files: 1) design descriptions as the LLM input, 2) test benches to verify the correctness of LLM-generated HDL code, and 3) the correct HDL code (i.e., reference model), typically handcrafted by designers, as a reference.

Benchmarks for RTL Code Generation			
Benchmarks	Open-sourced	link	Date
RTLLM [12, 154]	✓	https://github.com/hkust-zhiyao/rtllm	2023-10
VerilogEval [14]	✓	https://github.com/NVlabs/verilog-eval	2023-12
VerilogEval v2[165]			2024-08
CreativeEval [147]	✓	https://github.com/matthewdelorenzo/creativeval	2024-04
RTL-repo [153]	✓	https://github.com/AUCOHL/RTL-Repo	2024-05
VHDL-Eval [148]			2024-06
ChatGPTV [149]	✓	https://github.com/aichipdesign/chipgptv	2024-11

Table 4. Collection of benchmarks on LLMs for RTL generation in Section 5.1. VerilogEval [14] and its second version [165] share the same open-source link.

RTLLM [12] is one of the first benchmarks on design RTL generation based on natural language descriptions. It introduces a comprehensive evaluation framework with three progressive goals: syntax correctness, functionality correctness, and design quality (i.e., PPA metrics). The benchmark includes 30 diverse designs, ranging from simple arithmetic circuits to complex systems like a RISC CPU, and provides automated evaluation pipelines with natural language descriptions, testbenches, and human-crafted reference designs. RTLLM also adopts a self-planning prompt engineering technique, which significantly improves the performance of GPT-3.5 [88] by decomposing the RTL generation task into planning and code generation steps. The latest version (i.e., RTLLM 2.0) is available in OpenLLM-RTL [154] and expands the benchmark to 50 designs.

VerilogEval [14] is the other pioneering benchmark on RTL generation based on natural language descriptions. It comprises 156 problems sourced from HDLBits, covering a wide range of topics from combinational circuits to finite state machines. VerilogEval offers two types of problem descriptions: machine-generated (using LLMs) and human-curated, ensuring clarity and reducing ambiguity. The benchmark provides an automated testing environment using the ICARUS Verilog simulator and employs the pass@k metric to evaluate functional correctness. Additionally, VerilogEval explores supervised fine-tuning (SFT) with a synthetic dataset of 8,502 problem-code pairs, demonstrating that fine-tuning can enhance LLM performance, especially for models not originally trained on Verilog. Based on VerilogEval [14], VerilogEval v2 [165] evaluates the performance of new models and enhances the infrastructure and further discusses the importance of prompt engineering for RTL generation task.

In addition to the widely adopted RTLLM and VerilogEval benchmarks, there are several other benchmarks that assess LLMs in RTL code generation. CreativeEval [147] evaluates LLM creativity in Verilog generation based on fluency, flexibility, originality, and elaboration, finding GPT-3.5 to be the most creative among tested models. RTL-Repo [153] collects 4,000 GitHub samples to

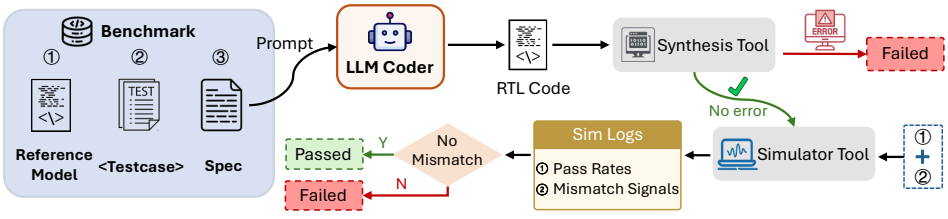


Fig. 11. Illustration of the benchmark on LLMs for RTL generation, recent works are covered in Section 5.1. Benchmark workflow comprises three steps: 1) LLMs generate RTL code from specifications, 2) The code is input into a synthesis tool to identify syntax errors, and 3) A simulation process checks for mismatches against predefined reference models or test case golden results.

LLM for RTL Code Generation				
Works	Open	Link	Date	Method
Nair et a. [158]*			2023-02	Prompt engineering
Enrique et al. [238]	✓	https://github.com/99EnriqueD/verilog_autocompletion	2023-04	
ChipGPT [133] (Opt)			2023-06	
RTLLM [12]	✓	https://github.com/hkust-zhiyao/rtllm	2023-08	
AutoChip [136]	✓	https://github.com/shailja-thakur/AutoChip	2023-11	
Chip-Chat [135]			2023-11	
Sandal et al. [161] (Opt)			2024-01	
Goh et al. [164]			2024-03	
VerilogCoder [144]			2024-08	
AIVRIL2 [138]			2024-09	
Nakkab et al. [159]			2024-09	
Vijayaraghavan et al. [160]			2024-09	
MAGE [152]			2024-12	
RTLSquad [151]	✓	https://github.com/observerw/RTLSquad	2025-01	
VRank [239]			2025-01	
DeLorenzo et al. [162] (Opt)			2024-02	Monte Carlo tree search
DAVE [132]	✓		2020-11	SFT with private data
VerilogEval [14]	✓	https://github.com/NVlabs/verilog-eval	2023-09	
ChipNemo [13]			2023-10	
BetterV [141] (Opt)			2024-02	
Chang et al. [142]			2024-05	
VeriSeek[163]	✓	https://huggingface.co/LLM-EDA/VeriSeek	2024-08	SFT with RL
DeepRTL [156]			2025-02	Representation Learning
VeriGen [137]	✓	https://github.com/shailja-thakur/vgen	2023-07	UFT
RTLCoder [145]	✓	https://github.com/hkust-zhiyao/RTL-Coder	2023-12	SFT with open-sourced data
CodeV [139]	✓	https://github.com/IPRC-DIP/CodeV	2024-07	
MG-Verilog [146]	✓	https://github.com/GATECH-EIC/mg-verilog	2024-07	
AutoVCoder [140]	✓	https://github.com/sjtu-zhao-lab/AutoVCoder	2024-07	
Origen [143]	✓	https://github.com/pku-liang/OriGen	2024-09	
CraftRTL [157]	✓	https://github.com/NVlabs/CraftRTL	2024-09	

Table 5. Works on LLMs for RTL generation. In the 'Works' column, denotation '*' refers to works on security, while '(Opt)' means the work focuses on design optimization. Though VerilogEval [14, 165] is a benchmark (in Table 4), it also proposes SFT with problem-pair pairs, while the training dataset is not open-sourced. DeLorenzo et al. [162] introduce an RTL generation framework that integrates the MCTS sampling process, which we classify as a form of prompt engineering since it solely alters the inference process, without necessitating fine-tuning. Some works provide code through GitHub but don't provide open-source models or training datasets. We classify them as SFTs with private data.

evaluate LLMs on *'long-range dependency handling'* ability through several metrics. However, the dataset lacks corresponding functional specifications, preventing it from being considered a standard benchmark. VHDL-Eval [148] addresses the lack of VHDL-specific benchmarks, offering 202 problems with self-verifying testbenches to evaluate functional correctness through zero-shot generation and fine-tuning. ChatGPTV [149] introduces a multi-modal benchmark for Verilog synthesis, incorporating visual inputs to improve LLM performance in handling spatial circuit complexity, showing significant accuracy gains over text-only approaches.

5.1.2 *RTL code generation techniques.*

RTL code generation is the most extensively explored application of LLM-assisted EDA techniques. Given the growing body of work in this area, we categorize existing approaches into four distinct strategies. Table 5 and Figure 12 list the comparison and timeline of all related works, respectively.

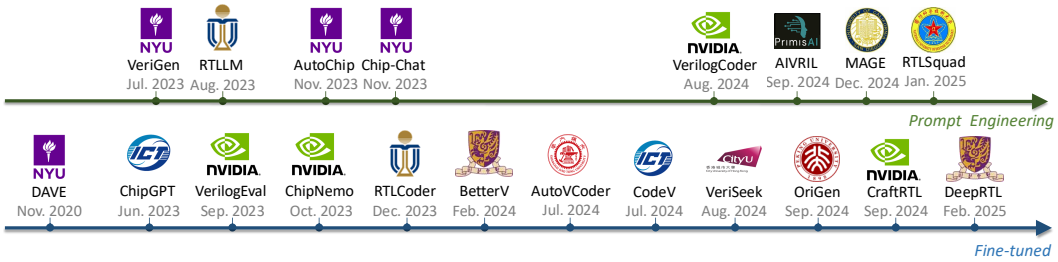


Fig. 12. Timeline of works on RTL generation, covered in Section 5.1. The timeline also includes the works of generation for design optimization (e.g., BetterV [141]). Recent works (e.g., Origen [143] and VerilogCoder [144]) show a trend of utilizing feedback from EDA tools to improve generation quality.

(1) **Prompt engineering.** This approach designs specific prompts to guide LLMs in generating RTL outputs. The effectiveness of the generated code largely hinges on the quality of these prompts, which often requires iterative refinements and experimental trial-and-error. Well-crafted prompts can lead to correct and even high-quality RTL code.
(2) **LLMs trained on *private* datasets with *instruction-code pairs*.** This approach fine-tunes LLMs (based on already pre-trained LLMs) using proprietary data, such as industrial in-house circuit designs. It tailors models to an organization's needs, enhancing performance but requires significant resources and access to high-quality private data.
(3) **LLMs trained on *open* datasets with *code only*.** This approach uses open-source codebases to fine-tune LLMs, eliminating the need for labor-intensive, high-quality datasets that require manual annotation. Such unsupervised fine-tuning process can help LLMs capture inherent structures of RTL code but is less effective for instruction-following tasks.
(4) **LLMs trained on *open* datasets with *instruction-code pairs*.** Fine-tunes models on pairs of design specifications and RTL implementations, helping them translate specifications into code. This requires a large number of high-quality pairs, which can be challenging to obtain but is quite effective in boosting LLMs' ability on hardware code generation tasks.

The first strategy, prompt engineering, primarily leverages commercial LLMs via API calls. In contrast, the other three strategies focus on customizing local LLMs by fine-tuning pre-trained models, mostly from open-source communities. Each strategy has its own advantages and drawbacks. Commercial models (e.g., GPT) reduce the substantial costs of training and deploying LLMs but may raise security and intellectual property concerns. On the other hand, fine-tuning local open-sourced LLMs (e.g., Llama, DeepSeek) can address these security and IP issues but requires significant resources and limits the model size. Smaller-scale customized LLMs tend to be less general compared with large commercial solutions.

The four distinct strategies for RTL generation using LLMs present unique advantages and challenges. **Prompt engineering** focuses on crafting precise prompts to guide LLMs in generating RTL code, which usually involves iterative refinement with EDA tools. In contrast, **LLMs trained on private datasets** ('*Supervised Fine-Tuning*', denoted as 'SFT' in Table 5) enhance model performance by fine-tuning them with proprietary data tailored to specific organizational needs. However, this method demands computational resources for fine-tuning, and the private circuit dataset is not open-sourced to facilitate the advancement of the community. Alternatively, **LLMs trained on open datasets with code only** ('*Unsupervised Fine-Tuning*', denoted as 'UFT' in Table 5) leverage open-source codebases for unsupervised fine-tuning, reducing the need for labor-intensive dataset preparation. However, this approach is less effective for RTL generation, which requires strict adherence to specific instructions including design descriptions. The UFT trains LLMs to predict

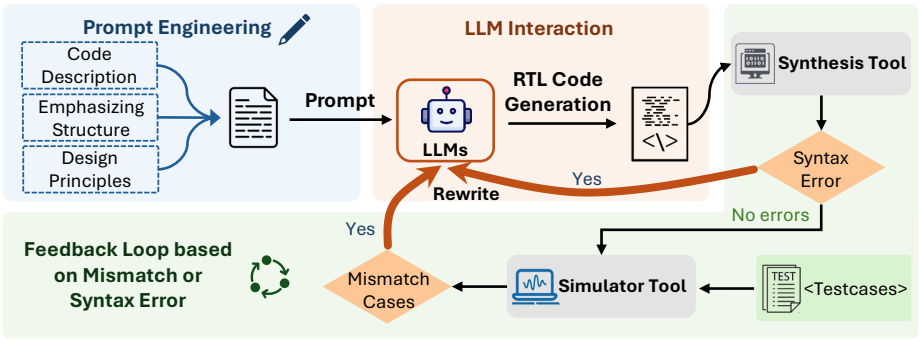


Fig. 13. Illustration of the basic flow of RTL design using prompt engineering. Related works are covered in Section 5.1. The design specification will be combined with manually designed structure analysis and design principles (in VerilogCoder [144]), and then the LLMs will take the prompt to generate corresponding RTL code. Most early works (RTLLM [12], AutoVCoder [140]) on RTL generation stop at this step. The more recent works incorporate the feedback from EDA tools into the design flow, for example, utilizing the error information and mismatch log to prompt LLMs for rewriting.

the next token based on the preceding context, which aids LLMs in grasping code syntax, but is limited in understanding required functional specifications. Consequently, this UFT approach is typically less effective and thus less adopted. Finally, **LLMs trained on open datasets with instruction-code pairs** (these methods also involve *Supervised Fine-Tuning*, denoted as 'SFT') utilize pairs of design specifications and RTL code to train the LLMs, and the dataset is open-sourced. This last strategy is both effective and benefits the community with open-source datasets.

Figure 14 compares the performance of various models on VerilogEval-Human [14] and RTLLM [12] over time, including both general-purpose coding LLMs (e.g., DeepSeek-Coder, CodeLlama) and RTL-specific coding LLMs (e.g., RTL-Coder [145], BetterV [141]). This comparison highlights the advancements in Verilog code generation, illustrating how domain-specific fine-tuning and architectural modifications enhance the effectiveness of LLMs in hardware design automation.

Strategy 1: Prompt engineering. Prompt engineering is one of the earliest strategies used for RTL generation due to its simplicity and effectiveness in applying LLMs to circuit design, including commercial LLMs. While recent efforts have focused on developing new fine-tuned LLMs for RTL generation, research about prompt engineering continues to evolve. Prompt engineering [240] mainly focuses on designing and optimizing input prompts to effectively communicate with LLMs and elicit desired design generations. The advantage of this methodology is the elimination of the requirement for fine-tuning and adaptability across different LLMs. Tons of works [12, 133, 135, 136, 138, 158, 159, 238] have explored applying and customizing advanced prompt engineering techniques for RTL code generation by LLMs, as demonstrated in Figure 13.

Chip-Chat [135], as a pioneering work, investigates the use of conversational LLMs, such as OpenAI's ChatGPT, in translating natural language specifications into HDLs for circuit design. Through a case study, the authors explore the collaborative design of an 8-bit accumulator-based microprocessor with GPT-4. The methodology involves breaking down the design into subtasks managed through conversation threads, where GPT-4 generates Verilog code guided by a human engineer who verifies and refines the output. The study finds that while LLMs can produce high-quality code and act as effective design assistants, they require human oversight for specification corrections and struggle with verification tasks. This research highlights the potential of LLMs to enhance productivity in circuit design when used as a complement to human expertise.

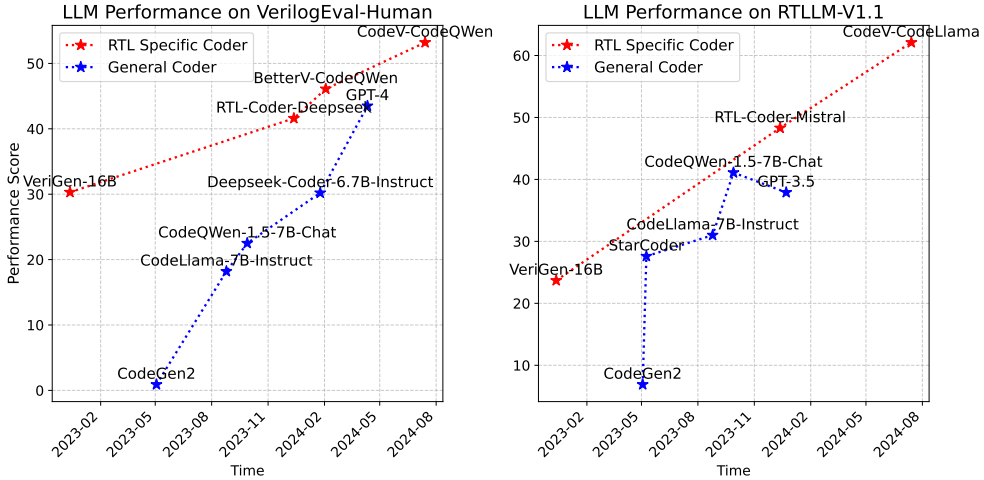


Fig. 14. Performance comparison of various approaches to RTL generation tasks. Involved works are covered in Section 5.1. The first figure illustrates the performance of different methods over time as evaluated by VerilogEval [14], while the second figure presents the results obtained from RTLLM [12].

Recently, VerilogCoder [144] presents a novel framework utilizing multiple AI agents to automate Verilog code generation and correction. It introduces a task and circuit relation graph for structured task decomposition, ensuring the inclusion of essential signal and state transition details. The system incorporates an AST-based waveform tracing tool for debugging, allowing agents to identify and correct functional errors. Using the ReAct [241] technique, agents iteratively interact with Verilog tools, including syntax checkers and simulators, to refine the code. This methodology significantly enhances the automation of circuit design.

Strategy 2: LLMs trained on private dataset with instruction-code pairs. Aside from prompt engineering for circuit design generation, another powerful technique is fine-tuning. Fine-tuning can be categorized into unsupervised fine-tuning and supervised fine-tuning. Here we focus on models that utilize supervised fine-tuning on private datasets. Many solutions in this domain are developed by industrial companies or in collaboration with industrial organizations. As a result, the models or datasets are often not open-sourced. DAVE [132] presents a pioneering custom dataset generation process that employs a template-based approach to create instruction-code pairs. They frame the Verilog generation task as a machine translation problem, fine-tuning a GPT-2 model to produce Verilog code from English descriptions. The training dataset generation process utilizes "Task/Result metastructure" that outlines the type of digital design task and relevant details, together with templates representing various scenarios such as combinational assignments and registers. The generated dataset is not open-sourced but includes diverse task instances to aid in fine-tuning the model. AutoVCoder [140] is a framework designed to improve the accuracy of LLMs in generating Verilog code. It addresses the challenges of low syntactic and functional correctness in LLM-generated RTL code by employing three key techniques: a high-quality hardware dataset generation method, a two-round LLM fine-tuning process, and a domain-specific RAG mechanism. The framework uses a code scorer to filter a large dataset of Verilog code from GitHub and generates a synthetic dataset using GPT-3.5. The two-round fine-tuning leverages these datasets, and the RAG module is designed to enhance the process by providing relevant context during code generation. OriGen [143] is a new open-source framework for generating RTL Verilog code. It addresses the limitations of existing open-source LLMs by incorporating a novel code-to-code augmentation technique and a self-reflection mechanism. The augmentation method uses a commercial LLM

(Claude3-Haiku) as a "teacher" to improve the quality of open-source RTL datasets. To address the scarcity of high-quality Verilog data, CodeV [139] leverages the observation that LLMs excel at summarizing Verilog code, rather than generating it from scratch. The system operates by first collecting and filtering a large corpus of high-quality Verilog modules from open-source repositories. These modules are then fed into GPT-3.5, which generates multi-level summaries—detailed functional descriptions and higher-level problem statements—for each module. These description-code pairs form a high-quality dataset used to fine-tune base LLMs (CodeLlama, DeepSeekCoder, and CodeQwen), resulting in the CodeV series of models.

DeepRTL [156] introduces a unified representation model to enhance both understanding and generation of Verilog code. The model addresses limitations in previous approaches, which focus primarily on Verilog code generation, neglecting the critical task of understanding. DeepRTL [156] is fine-tuned on a comprehensive dataset that aligns Verilog code with multi-level natural language descriptions, covering line, block, and module levels with both detailed and high-level functional descriptions. The dataset includes both open-source and proprietary Verilog code, annotated using a CoT approach with GPT-4 and verified by human experts. The authors introduce a novel benchmark for Verilog understanding and propose using semantic evaluation metrics like embedding similarity and GPT score, which capture semantic coherence more effectively than traditional methods like BLEU and ROUGE. Additionally, the paper employs curriculum learning, allowing the model to incrementally build knowledge from simpler to more complex tasks, enhancing its performance in both understanding and generation of Verilog code.

Strategy 3: LLMs trained on open datasets with code only. In the initial phase of fine-tuning LLMs for RTL design generation, early efforts primarily relied on *unsupervised* data sourced from platforms like GitHub and other open-source code repositories. The key benefit of using unsupervised datasets is their ease of training and the large volume of existing unlabeled data, eliminating the need for labor-intensive labeling tasks. However, the limited label alignment of these datasets limits their effectiveness in training LLMs for RTL generation. These techniques mainly enable LLMs to understand language patterns, structures, and semantics by processing vast amounts of text data. LLMs are tasked to predict the next token given the previous context [88].

For example, VGen [15], as the pioneering work, first evaluates the ability of unsupervised LLMs to generate Verilog code, a critical aspect of circuit design. The authors fine-tune several pre-trained LLMs on a large dataset of Verilog code collected from GitHub and textbooks, creating the largest training corpus for this purpose. They develop an evaluation framework that includes test benches for assessing both the syntactic and functional correctness of the generated code across various problem scenarios. Wang et al. [163] explores the use of LLMs for automatically generating Verilog code from natural language specifications. It introduces a novel approach that employs reinforcement learning with golden code feedback, specifically using proximal policy optimization and a reward function based on the similarity of abstract syntax trees between generated and reference code. This method enhances the semantic evaluation of the generated code and addresses the limitations of existing open-source models, which often lack performance compared to commercial alternatives. The authors present their model, VeriSeek, which has 6.7 billion parameters and achieves state-of-the-art results.

Strategy 4: LLMs trained on open datasets with instruction-code pairs. This process, known as '*instruction fine-tuning*' or '*supervised fine-tuning' [242]*, adjusts the model to follow specific instructions or prompts more effectively. Different from adopting in-house private datasets in strategy 2, this strategy tries to provide open-source datasets to benefit the community. To address the scarcity of high-quality training data, RTLCoder [145] introduces an automated dataset generation workflow. As illustrated in Figure15, the dataset generation workflow follows a three-step process, establishing a foundational paradigm for dataset creation. Its generated dataset enables

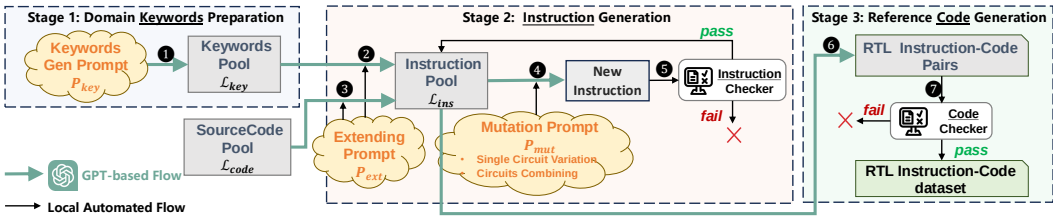


Fig. 15. Data generation flow of RTLCoder [145]. Following works (like AutoVCoder [140], Origen [143]) also adopts similar methodologies for dataset generation. The dataset generation consists of three steps. The first two steps are designed to generate diverse instructions (design specifications) and the final step is to generate high-quality reference code for the next fine-tuning step.

a fine-tuned LLM coder that outperforms GPT-3.5 [88] and achieves performance comparable to GPT-4 [88]. This pioneering work provides the first open-source RTL LLM coder with instruction fine-tuning. Similar to the paradigm introduced by RTLCoder, CraftRTL [157] further analyzes existing LLM's performance on Verilog, identifying two key weaknesses: poor handling of non-textual representations (like Karnaugh maps and waveforms) and inconsistent performance due to minor coding errors. Targeting these weaknesses, CraftRTL creates a "correct-by-construction" synthetic dataset that includes Karnaugh maps, finite state machines, and waveform representations. They develop an automated framework for generating detailed error reports that identify common minor mistakes in code completions, which are then used to create a targeted code repair dataset by injecting errors into correct open-source code.

5.2 LLMs for HLS Code Generation

Similar to *RTL code* generation, several works have explored LLM-based *HLS code* generation to improve automation and efficiency when designing hardware with high-level programming languages. As summarized in Table 6, existing solutions primarily rely on prompt engineering without fine-tuning the models. Additionally, many works are open-sourced. Notably, HLSPilot [167] and Liao et al. [170] introduce new benchmarks for evaluating HLS code generation performance.

For example, SynthAI [169] introduces a multi-agent generative AI framework for modular HLS design, integrating ReAct agents, CoT prompting, RAG, and web search capabilities to enhance decision-making. By systematically planning and executing modular designs, SynthAI improves design quality and scalability. HLSPilot [167] focuses on hybrid CPU-FPGA architectures, proposing a three-stage approach: C/C++ to HLS translation, design space exploration, and LLM-based profiling. It integrates C-to-HLS optimization strategies to generate complex circuit designs, employs a DSE tool for pragma parameter tuning, and leverages LLMs for performance profiling to identify bottlenecks and optimize HLS designs. Liao et al. [170] investigate the translation of natural language specifications or C code into RTL, evaluating the capability of LLMs to automate hardware design. C2HLSC [168] explores fully automated C-to-HLS transformation, refactoring generic C code into an HLS-compatible format while supporting hierarchical designs and pragma generation for optimizing area and throughput. These works collectively highlight the potential of LLMs in improving HLS design automation, enabling more efficient translation from high-level code to synthesizable hardware descriptions.

5.3 LLMs for Design Optimizations

During circuit code generation using LLMs, besides functional correctness focused by Section 5.1 and 5.2, design quality metrics such as power, performance, and area (PPA) are also critical for ensuring

LLM for HLS Generation							
Method	New Model	New Dataset	Open Method	New Benchmark	Prompt Engineering	Link	Date
SynthAI [169]			✓		✓	https://github.com/sarashs/FPGA_AGI	2024-05
HLSPilot [167]			✓	✓	✓	https://github.com/xcw-1010/HLSPilot	2024-08
Liao et al. [170]				✓	✓		2024-08
C2HLSC [168]			✓		✓	https://github.com/Lucaz97/c2hlsc	2024-11

Table 6. Existing explorations in LLM-aided HLS code generation, covered in Section 5.2.

LLM for Design Optimization			
Works	Open	Link	Date
Martine et al. [173]			2023-07
Sandal et al. [161]			2024-01
BetterV [141]			2024-02
DeLorenzo et al. [162]			2024-02
RTLRewriter [172]	✓	https://github.com/yaoxufeng/RTLRewriter-Bench	2024-09
Xu et al. [174]			2024-10

Table 7. Collection of works on design optimization, covered in Section 5.3.

efficiency and practicality. Recent advancements have explored optimizing LLM-generated circuits, focusing on both **RTL code optimization** and **HLS code optimization** to enhance hardware performance. A detailed comparison of existing works in this domain is provided in Table 7.

RTL code optimization. RTL code optimization leverages LLMs to refine hardware designs for better efficiency, focusing on improving PPA metrics. For example, ChipGPT [133] employs an enumerative search strategy, generating multiple design variations and selecting the one with the best PPA. BetterV [141] fine-tunes LLMs on domain-specific Verilog datasets, applying instruct-tuning and generative discriminators to improve Verilog code quality and optimize synthesis outcomes. However, current evaluations in BetterV primarily assess design quality based on AIG node reduction during synthesis, without directly considering final PPA metrics. RTLRewriter [172] introduces a framework for RTL code rewriting, breaking down large circuits into smaller segments to enhance synthesis efficiency and leveraging multi-modal program analysis to incorporate visual and textual information. Its benchmark demonstrates superior performance compared to traditional RTL compilers such as Yosys and E-graph. Additionally, the work by Martínez et al. [173] focuses on identifying key computational patterns like GEMM, convolution, and FFT within hardware code using LLM-based prompting techniques. Their method reduces false positives by employing a two-phase prompting approach, first interpreting the code and then verifying algorithm presence, highlighting the importance of prompt engineering for optimizing LLM-driven hardware design.

HLS code optimization. Besides RTL code optimization, optimizing HLS code, particularly pragma optimization, is a crucial task in high-level synthesis. Xu et al. [174] propose RALAD (Retrieve Augmented Large Language Model Aided Design), a framework leveraging LLMs and RAG to optimize HLS programs without requiring computationally expensive fine-tuning. HLS allows circuit design using high-level languages like C/C++, but manual optimization remains highly expertise-driven. RALAD mitigates this challenge by embedding user code and a knowledge base (e.g., FPGA textbooks), retrieving relevant code snippets via a top-k search, generating prompts that incorporate user instructions and retrieved snippets, and using an LLM like CodeLlama to produce optimized code. The study also explores the impact of manual annotations to further refine optimization quality, demonstrating the framework's effectiveness in automating HLS code improvements.

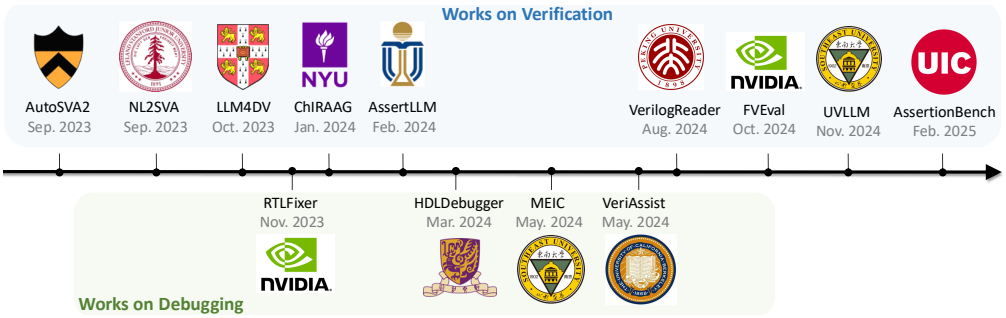


Fig. 16. Timeline of RTL verification (Section 5.4) and debugging works (Section 5.5).

5.4 LLM for Hardware Code Verification

In addition to circuit code generation, verifying the functional correctness of circuit designs is a critical yet highly labor-intensive task that heavily relies on human engineers. To address this challenge, LLM-based solutions have been explored to automate hardware verification. Table 8 summarizes existing works in this direction. Current LLM-based verification approaches focus on two primary directions: 1) **Assertion generation with LLMs.** These approaches leverage LLMs to generate assertions based on design specifications or RTL code [177, 178, 181, 182, 188–191]. The generated assertions are then used to validate whether the design under test (DUT) complies with its specifications, with either formal verification tools (e.g., Cadence JasperGold) for static formal property verification or simulation tools (e.g., Synopsys VCS) for dynamic verification on test benches. 2) **Test bench generation with LLMs.** LLMs are also employed to generate test stimuli, enhancing the simulation-based verification process [183–185, 191, 195]. The comparison of these explorations is listed in Table 8, with the timeline demonstrated in Figure 16, almost all existing explorations directly employ prompt engineering due to the lack of high-quality verification data for fine-tuning. Some of these verification efforts focus specifically on security verification, which will be further discussed in Section 5.6.

5.4.1 Assertion generation with LLMs.

We categorize existing works on assertion generation into two main types: assertion generation benchmarks and assertion generation techniques. The former focuses on evaluating the effectiveness of LLMs in generating functionally correct assertions, while the latter explores various methodologies to improve the accuracy and reliability of LLM-generated assertions. We detail these two categories below.

Benchmarking LLM-aided assertion generation. Similar to RTL code generation, benchmarking is crucial for evaluating the quality of LLM-generated assertions. The evaluation process involves three key aspects: syntax correctness, functional correctness, and overall assertion quality. Syntax correctness can be verified using RTL code compilers, while functional correctness can be validated through simulation-based verification or formal property checking based on the golden RTL implementations. However, assessing assertion quality remains an open challenge, as it depends on multiple factors, such as completeness and relevance to the design specification.

Currently, key assertion generation benchmarks include AssertionBench [187], AssertEval [154], and FVEval [186], all of which use Cadence JasperGold for formal property verification of generated assertions against golden RTL implementations. Specifically, AssertionBench [187] consists of 100 Verilog hardware designs from OpenCores [120], with formally verified assertions derived from GOLDMINE [243] and HARM [244] tools. The evaluation metrics include syntax correctness and

LLM for RTL Verification							
Method	New Model	New Dataset	Open Model	Open Benchmark	Prompt Engineering	Link	Date
AutoSVA2 [177]					✓		2023-09
NL2SVA [189]					✓		2023-09
LLM4DV [184]				✓	✓	https://github.com/ZixiBenZhang/ml4dv	2023-10
ChIRAAG [182]					✓		2024-01
AssertLLM [181]			✓	✓	✓	https://github.com/hkust-zhiyao/AssertLLM	2024-02
Xiao et al. [194]					✓		2024-03
Blocklove et al. [196]					✓		2024-04
Liu et al. [190]	✓	✓					2024-04
Huang et al. [191]					✓		2024-05
Bhandari et al. [195]				✓	✓		2024-06
VerilogReader [185]					✓	github.com/magicYang1573/llm-hardware-test-generation	2024-06
FVEval [186]		✓		✓	✓	https://github.com/NVlabs/FVEval	2024-10
UVLLM [183]			✓		✓	https://github.com/amyuch/UVLLM	2024-11
AssertionBench [187]				✓	✓		2025-02

Table 8. Explorations in LLM-aided RTL code verification (Section 5.4).

functional correctness. AssertEval [154] from OpenLLM-RTL [154] includes 17 OpenCores [120] designs, each accompanied by a natural language specification and golden RTL implementation. It evaluates assertions based on syntax correctness, functional correctness, and COI (cone-of-influence) coverage. FVEval [186] assesses assertions in three scenarios: (1) NL2SVA-Human, generating assertions from human-written specifications and real-world testbenches; (2) NL2SVA-Machine, translating formal logic from synthetic natural language descriptions to SystemVerilog assertions; and (3) Design2SVA, directly generating assertions from RTL designs. It evaluates various LLMs (e.g., GPT-4o, Gemini, LLaMA3) based on syntax correctness, full functional correctness, and partial correctness (assertions that are logically related but not fully equivalent to the reference).

Generate design assertions with LLMs. LLMs automate hardware verification by leveraging natural language specifications and RTL code to produce SystemVerilog Assertions (SVA). These techniques can be categorized based on their input types: (1) Natural language specifications alone (e.g., ChIRAAG [182], AssertLLM [181]). (2) RTL code alone (e.g., AutoSVA2 [177]). (3) Both specification and RTL code (e.g., NL2SVA [189]). Due to the scarcity of high-quality assertion datasets, most works employ prompt engineering rather than fine-tuning LLMs. Evaluation methods typically rely on formal property verification (FPV) using tools like Cadence JasperGold, ensuring that generated assertions maintain logical correctness. Some works, such as ChIRAAG [182], also incorporate simulation-based validation using Synopsys VCS with test benches.

For example, AutoSVA2 [177] prompts GPT-4 with RTL code and a refined rule-based system to generate valid SVAs, validated through FPV. NL2SVA [189] employs few-shot prompting with both RTL and natural language descriptions to guide assertion generation, also evaluated via FPV. ChIRAAG [182] relies solely on natural language specifications, using prompt engineering for assertion synthesis, with validation conducted through simulation. AssertLLM [181] processes entire specification documents, utilizing a three-phase approach where different LLMs handle specification extraction, waveform analysis, and assertion generation, with verification performed through FPV.

5.4.2 Test bench generation with LLMs.

In addition to assertion generation, recent advancements in LLM-based verification have introduced automated test bench generation [183–185, 191, 195], significantly reducing the manual effort involved in verifying RTL designs. These approaches aim to enhance coverage metrics, including code coverage and functional coverage, by generating high-quality test benches for simulation.

Existing explorations also fall into these main categories: (1) Test bench generation for *code coverage*. This type focuses on measuring how thoroughly the RTL code is exercised during simulation. This includes metrics such as statement coverage, branch coverage, toggle coverage, and FSM state coverage. Achieving high code coverage ensures that most structural elements of the design have been tested but does not guarantee full functional correctness. (2) Test bench generation for *functional coverage*. This type ensures that all intended design behaviors are tested according to the specification. Functional coverage is often defined using assertions and covergroups, verifying that different functional scenarios, corner cases, and expected behaviors are exercised. Unlike code coverage, functional coverage validates the correctness of the design beyond just its structural execution. We detail the two categories below.

Test bench generation for code coverage. VerilogReader [185] integrates LLMs into coverage-directed test generation, focusing on achieving code coverage closure by generating test stimuli that target uncovered RTL lines and branches. It takes as input a Verilog design under test (DUT), natural language descriptions, and code coverage reports from simulations. The output consists of automatically generated test stimuli designed to improve code coverage in RTL verification. To generate test inputs effectively, VerilogReader employs prompt engineering with a Prompt Generator that structures LLM interactions in two stages: first, understanding the DUT and its current coverage status, and second, generating test inputs in a structured JSON format. Additionally, it includes a Coverage Explainer, which transforms raw simulator coverage reports into an LLM-readable format, and a DUT Explainer, which enhances LLM comprehension of Verilog code by providing natural language descriptions and test guidance.

Test bench generation for functional coverage. Most existing explorations [183, 184, 191, 195] focus on functional coverage, as LLMs more excel in understanding RTL functionality and specifications rather than analyzing RTL code structure, which is required for code coverage. For example, VeriAssist [191] takes the design specification as input and generates initial RTL code along with corresponding test cases. It employs a self-verification process, where the generated RTL is simulated with test cases while considering timing constraints. This is followed by a self-correction mechanism, where the LLM refines the RTL design based on simulation feedback, addressing compilation and functional errors. By mimicking a human-in-the-loop design approach, VeriAssist [191] improves the accuracy and correctness of both RTL code and test benches. Another work UVLLM [183] integrates LLMs with Universal Verification Methodology (UVM) to automate test case generation and RTL code repair. The framework consists of four steps: pre-processing, where linters and LLMs eliminate syntax errors; UVM processing, which generates and runs test cases within a UVM testbench; post-processing, which analyzes simulation logs to identify errors; and repair, where LLMs generate RTL patches based on detected issues. While UVLLM is open-sourced and showcases LLMs' potential in verification automation, challenges remain, including the need for extensive training data and the high computational cost of large-scale LLM inference.

5.5 LLM for Hardware Code Debugging

Debugging in hardware design involves identifying and fixing both syntax and functional errors in circuit implementations. Traditionally, engineers conduct this process of fixing bugs manually, making it a tedious and labor-intensive task. Recent advancements in LLMs automate hardware debugging, reducing human intervention and improving efficiency. Existing research explores LLM-assisted debugging for both RTL and HLS code, offering new methodologies for error detection, root cause analysis, and automated patch generation, as detailed below.

5.5.1 LLM for RTL code debugging.

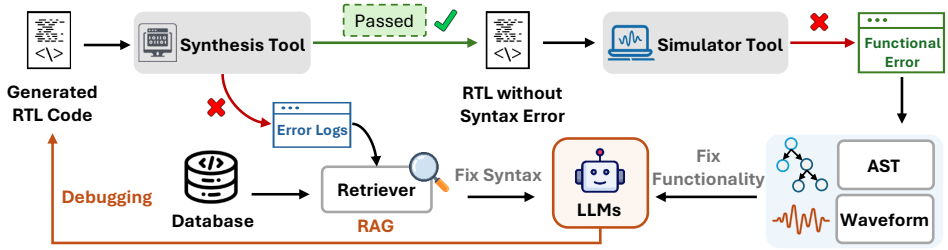


Fig. 17. Overview of the LLM-based flow for RTL debugging, recent works are covered in Section 5.5. This approach includes two input methods for LLM solutions: the first method assigns debugging tasks directly to the LLM without supplementary information, while the second method enhances the debugging process by incorporating error information from EDA tools. This error log can be input directly into the LLM or used to query a pre-defined debugging database (RAG). Additionally, some approaches utilize AST or waveform tracing tools to more effectively identify problematic code segments.

LLM for RTL Debugging							
Method	New Model	New Dataset	Open Model	Open Benchmark	Prompt Engineering	Link	Date
RTLFixer [176]			✓	✓	✓	https://github.com/NVlabs/RTLFixer	2023-11
HDLDebugger [200]					✓		2024-03
VeriAssist [191]							2024-05
MEIC [199]							2024-05
Qayyum et al. [204]							2024-06
VerilogCoder [144]					✓		2024-08
UVLLM [183]			✓		✓	https://github.com/amyuch/UVLLM	2024-11

Table 9. Explorations in LLM-aided RTL code debugging (Section 5.5).

RTL debugging focuses on resolving errors identified during the verification process. Unlike verification, which primarily detects inconsistencies, debugging involves both locating and correcting these issues to ensure functional correctness. For example, representative debugging works [176, 191, 199] leverage LLMs for both bug detection and bug fixing, emphasizing the automated correction of RTL errors. The debugging process typically consists of two key steps: (1) identifying the bug by pinpointing the exact error location within the RTL code and (2) fixing the bug by generating corrected RTL logic. While verification highlights potential failures, debugging requires deeper reasoning to determine the root cause of errors and propose appropriate fixes. Table 9 and Figure 16 demonstrate the comparison and timeline of these explorations, respectively.

RTL bugs are broadly categorized into syntax bugs and functional bugs [199]. Syntax bugs, such as missing semicolons or incorrect module instantiations, can be directly flagged by compilers (i.e., synthesis tools). Functional bugs, on the other hand, require executing test cases or formal verification to identify behavioral mismatches. Based on this classification, we categorize existing works in LLM-assisted RTL debugging into **syntax debugging** and **functional debugging**. Since functional debugging involves more complex reasoning and deeper analysis of design behavior, tools capable of addressing both syntax and functional errors are classified as functional debuggers.

Syntax debugging. RTLFixer [176] and HDLDebugger [200] are among the pioneering works to explore LLM-assisted RTL syntax debugging. Both works leverage the RAG technique to improve debugging accuracy by transforming syntax-buggy RTL code into syntax-correct RTL designs. RTLFixer [176] integrates RAG and ReAct prompting, creating an autonomous debugging agent that retrieves expert guidance and applies iterative reasoning to correct syntax errors effectively. It also introduces VerilogEval-Syntax, a debugging dataset consisting of 212 erroneous Verilog

implementations to benchmark LLM performance in syntax correction. HDLDebugger [200], developed around the same time, similarly employs RAG to retrieve relevant debugging information from documentation and code databases. Additionally, it incorporates a self-guided fine-tuning process to improve LLM-based debugging accuracy. The framework also includes a novel data generation module that synthetically creates pairs of buggy and corrected HDL code using a reverse engineering approach. Both methods significantly enhance LLM capabilities in syntax debugging by integrating retrieval-based contextual learning and structured reasoning techniques.

Functional debugging. Compared with syntax debugging, functional debugging is significantly more challenging as it requires deep reasoning about circuit functionality and identifying the root cause of errors. Recent works [144, 183, 191, 199, 204] have explored LLM-based approaches to address functional RTL debugging. VeriAssist [191] enhances pre-trained LLMs with self-verification and self-correction techniques. The framework generates test cases alongside RTL code and simulates the generated design to detect functional errors. If discrepancies are identified, self-correction mechanisms iteratively refine the RTL code based on simulation feedback, improving debugging accuracy. MEIC [199] proposes an LLM-based iterative debugging framework and introduces a new debugging benchmark based on RTLLM-v1.0 [12], an RTL generation dataset containing 15 source designs. By introducing 178 buggy variations of these designs, MEIC categorizes errors into syntax and functional bugs, providing a structured evaluation dataset for LLM-based debugging research. Qayyum et al. [204] integrate RAG into functional debugging by retrieving relevant RTL specifications and comparing them with the RTL implementation. This enables LLMs to detect inconsistencies and suggest fixes based on the intended circuit behavior, significantly improving debugging accuracy through formal specification guidance.

Beyond direct RTL code analysis, some works incorporate auxiliary sources such as abstract syntax tree (AST) and waveform analysis to enhance functional debugging. UVLLM [183] introduces an LLM-based unified verification methodology, leveraging AST representations to improve error localization and code corrections. Similarly, VerilogCoder [144] employs a rewriting mechanism that enhances debugging accuracy. This process integrates information from EDA tools, such as ASTs and waveform tracing tools, to refine LLM-driven RTL debugging. By combining LLM reasoning with structured program analysis, these methods offer improved robustness in identifying and correcting functional design errors.

5.5.2 LLM for HLS debugging.

Compared to RTL and HLS generation tasks, significantly fewer works focus on debugging at the HLS level. One of the pioneering efforts in this direction is Chrysalis [201], a benchmark designed for training and evaluating LLMs' capability to identify functional bugs in HLS code. Unlike syntax errors that can be easily detected by compilers, many functional bugs at the HLS level require deeper semantic analysis and program reasoning. Chrysalis provides a structured dataset that allows LLMs to learn patterns of common HLS-specific issues and evaluate their debugging performance in terms of both syntax correctness and functional accuracy. This benchmark sets the foundation for future research in LLM-assisted HLS debugging by offering a standardized dataset for evaluating model capabilities in detecting and resolving high-level synthesis errors.

5.6 LLMs for Hardware Security

Besides functional verification and debugging, a growing number of research explore the use of LLMs for hardware security verification and threat detection. Recent works [70–73, 179, 188, 193, 205–207, 213, 214] integrate LLMs into automated security analysis, detection of vulnerabilities, and protection of hardware designs. As summarized in Table 10, most of these approaches rely on prompt engineering to enhance security verification and threat detection.

LLM for Hardware Security							
Method	New Model	New Dataset	Open Model	Open Dataset	Open Benchmark	Prompt Engineering	Date
Pearce et al. [212]				✓		✓	2021-12
Baleegh et al. [203]						✓	2023-02
Kande et al. [188]					✓	✓	2023-06
DIVAS [179]						✓	2023-08
NSPG [178]	✓	✓					2023-08
SCAR [211]						✓	2023-10
Netlist Whisperer [210]						✓	2023-11
SecRT-LLM [205]		✓				✓	2024-05
Self-HWDebug [206]						✓	2024-05
Qayyum et al. [204]							2024-06

Table 10. Existing explorations in LLMs for security, covered in Section 5.6.

LLM-based research in hardware security can be divided into two primary directions. **Protective hardware security** focuses on using LLMs to detect vulnerabilities, generate security patches, and implement secure-by-design methodologies at the RTL and gate levels. These approaches aim to proactively mitigate security risks through automated analysis and verification techniques. **Offensive hardware security**, in contrast, explores how LLMs can facilitate attack strategies and identify potential hardware exploits. Together, these two research directions contribute to the development of more resilient defense mechanisms by providing insights into adversarial techniques and enabling the design of effective countermeasures. Below, we introduce representative works about both directions in detail.

LLM-aided protective hardware security. Research on LLM-assisted protective hardware security can be categorized into two key areas: ***security bug detection*** through security assertion generation and ***security bug fixing***, which involves identifying and debugging vulnerabilities. These approaches highlight the potential of LLMs in automating security analysis, enhancing verification processes, and mitigating hardware vulnerabilities.

In ***security bug identification***, LLMs have been explored to automate the detection of hardware vulnerabilities and generate security assertions. For instance, Kande et al. [188] demonstrate the potential of LLMs in generating hardware security assertions, a task that traditionally requires significant expertise. Similar to the functional assertion generation process, their framework employs LLM to generate security assertions based on security specifications and evaluates LLM performance using a benchmark suite of real-world designs and corresponding golden reference security assertions, analyzing the impact of prompt detail on accuracy. DIVAS [179] introduces an LLM-powered framework that automates SoC security analysis and policy-based protection. The system maps vulnerabilities to Common Weakness Enumerations (CWEs), generates verifiable SVAs, and implements security policies through security modules or wrappers. Evaluated on open-source benchmarks, DIVAS demonstrates effectiveness in automating SoC security analysis, policy enforcement, and vulnerability detection using the DiSPEL tool. Similarly, SecRL-LLM [205] propose a database containing 10,000 vulnerable finite state machine designs incorporating 16 security weaknesses. They further develop an LLM-based framework, integrating in-context learning and fidelity-check mechanisms to enhance both vulnerability insertion and detection in hardware designs.

Beyond vulnerability detection, some works also explore ***security bug fixing*** by leveraging LLMs for automated debugging and countermeasure implementation. Pearce et al. [212] conducted one of the earliest studies in this area, prompting LLMs to automatically repair software security vulnerabilities as early as 2021, prior to the emergence of today's more powerful models. Their work

LLM for Design Flow Automation			
Method	Open	Link	Date
ChatEDA [216]	✓	https://github.com/wuhy68/ChatEDAv1	2023-08
RAG-EDA [228]	✓	https://github.com/lesliepy99/RAG-EDA	2024-07
ChipAlign [221]			2024-12

LLM for Layout Design			
Method	Open	Link	Date
Ho et al. [224]			2024-05
FabGPT [222]			2024-07
Chen et al. [223]			2024-08
DRC-Coder [220]			2024-11

Table 11. Explorations on LLMs for design flow automation and layout design, covered in Section 5.7.

presents a comprehensive empirical evaluation of multiple commercial and open-source LLMs, including OpenAI Codex, AI21's Jurassic models, Polycoder, and gpt2-csrc, assessing their ability to generate secure and functional patches for synthetic, handcrafted, and real-world security bugs. Recently, Self-HWDebug [206] introduces a framework leveraging LLMs to generate debugging instructions for security issues. By defining a set of CWEs and corresponding mitigation strategies, the framework enhances LLM prompt effectiveness, enabling automated security debugging and vulnerability resolution. In the domain of side-channel attack (SCA) mitigation, Netlist Whisperer [210] and SCAR [211] propose LLM-driven solutions to enhance security at the hardware level. Netlist Whisperer [210] adopts a two-phase, pre-silicon LLM-based approach: first, a GPT-3 model identifies power leakage-inducing nets in a circuit, and then a second GPT-3 model generates an SCA-resistant netlist, eliminating the need for traditional power trace collection. SCAR [211] focuses on cryptographic accelerators, utilizing control-data flow graphs to identify and localize SCA vulnerabilities. It then employs a deep-learning explainer to analyze the vulnerabilities and leverages an LLM to automatically generate and insert security patches into the RTL code.

LLM-aided offensive hardware security. Besides protective solutions, research also explores how LLMs can be leveraged to execute security threats, such as automated hardware trojan insertion. For example, Kokolakis et al. [213] propose an LLM-based framework for automating hardware trojan insertion and evaluating its impact on a modern RISC-V microarchitecture. Their method begins with a filtering process to identify suitable modules for Trojan insertion. The RTL code of selected modules is provided to the LLM, which assists in implanting hardware trojans by modifying the design. While this approach demonstrates the feasibility of hardware trojan insertion using fine-tuned LLMs, further research is needed for more complex attack scenarios.

5.7 LLM for Design Flow Automation and Layout Design

5.7.1 LLM for design flow automation.

LLMs have also been explored for automating design flow processes, primarily in two key areas: ***design flow script synthesis*** [216] and ***chip-related question-answering*** (QA) [221, 228]. These applications aim to reduce human effort in configuring and optimizing EDA toolchains while improving accessibility to chip design knowledge. We compare existing works in Table 11.

For ***design flow script synthesis***, ChatEDA [216] is a pioneering work that leverages LLMs for automating EDA design flow execution. The framework decomposes user requests into structured sub-tasks, generates EDA scripts, and autonomously executes them using EDA tools. To enhance the model's understanding of EDA workflows, instruction tuning techniques are applied. Additionally, ChatEDA introduces a benchmark suite comprising 50 tasks that include simple flow calls, complex multi-step flow executions, and parameter-tuning scenarios. The evaluation process assesses both

the correctness and executability of the generated scripts using real EDA tools, followed by manual scoring to ensure logical coherence and practical usability.

For ***chip-related question-answering***, RAG-EDA [228] presents a customized RAG framework for EDA tool documentation QA. The framework processes EDA tool documentation and user queries, employing hybrid information retrieval methods that combine lexical search (i.e., TF-IDF, BM25) and semantic retrieval (i.e., vector embeddings). A contrastive learning-based reranker is trained to filter relevant documents, improving retrieval accuracy. The LLM is then fine-tuned through a two-stage approach: (1) domain knowledge pretraining using EDA textbooks and (2) instruction tuning with QA datasets. The evaluation metrics include retrieval recall (recall@k) for the retriever and reranker, and BLEU, ROUGE-L, and UniEval scores to assess the accuracy and factual consistency of generated answers. Additionally, ChipAlign [221] extends LLM capabilities for chip-related QA tasks by addressing the challenge of aligning domain-adapted large language models for chip design with strong instruction-following abilities. It proposes a training-free model merging approach that combines a domain-specific chip LLM with a general instruction-aligned LLM. Instead of retraining on instruction-following data, ChipAlign employs a novel geodesic interpolation technique in the weight space to produce a merged model that maintains chip design expertise while significantly improving instruction alignment.

5.7.2 LLM for layout design.

Some recent works employ foundation models for circuit layouts to enhance the **physical design process and manufacturability**, as demonstrated in Table 11. Since circuit layouts are typically represented in the format like images, these works typically integrate vision models with LLMs to better understand and process circuit layouts.

For instance, FabGPT [222] introduces a large multimodal model designed for wafer defect knowledge querying in semiconductor fabrication. It processes Scanning Electron Microscope (SEM) images of wafers alongside textual metadata extracted using Optical Character Recognition (OCR) and predefined label sets. By fusing visual and textual information, the model enhances defect detection and knowledge retrieval. A pre-trained multimodal encoder captures critical wafer defect features, while a prediction module identifies defect types. Additionally, the model incorporates a Q&A system with a modulation module that aligns visual and textual representations to improve interpretability in querying fabrication processes. Ho et al. [224] propose an LLM-based optimization framework for standard cell layout design, incrementally generating clustering constraints to enhance PPA and routability. Their study assesses existing LLMs' understanding of SPICE netlists, clustering constraints, and physical layout descriptions. Leveraging ReAct prompting, the model iteratively refines clustering decisions, improving standard cell layout optimization. Chen et al. [223] integrate reinforcement learning (RL) for OPC recipe optimization with a multi-modal LLM-backed agent system for recipe summarization. The RL component fine-tunes OPC parameters such as edge placement error (EPE) measurement points and polygon fragmentation to improve lithography accuracy. Meanwhile, the LLM-based agent extracts features, summarizes results, and generates structured OPC recipes, enhancing automation in semiconductor manufacturing. DRC-Coder [220] presents a multi-agent framework for automating design rule checking (DRC) code generation using LLMs and vision-language models. It mimics human DRC coding by decomposing tasks into interpretation and coding, assigning two specialized LLM agents to reduce hallucinations and enhance reasoning accuracy. The framework also integrates domain-specific functions, including foundry rule analysis, layout design rule violation (DRV) analysis, and automated debugging loops to refine DRC rule generation. By incorporating vision models, DRC-Coder can interpret design rule illustrations and layout structures, ensuring accurate and executable DRC scripts.

LLM for Architecture Design			
Method	Open	Link	Date
Yan et al. [226]			2023-06
Liang et al. [227]			2023-07
GPTAIGChip [134]			2023-09
SpecLLM [225]	✓	https://github.com/hkust-zhiyao/SpecLLM	2024-01

Table 12. Explorations in LLM-aided architecture design, covered in Section 5.8

LLM for Analog Circuit Design			
Method	Open	Link	Date
LADAC [229]			2023-12
AnalogCoder [230]	✓	https://github.com/anonyanalog/AnalogCoder	2024-05
FLAG [231]			2024-05
ADO-LLM [232]			2024-06
LaMAGIC [233]			2024-07
Artisan [234]			2024-11
LEDRO [235]			2024-11
AnalogXpert [236]			2024-12
AnalogGenie [237]	✓	https://github.com/xz-group/AnalogGenie	2025-01

Table 13. Works on LLMs for analog circuit design, covered in Section 5.9.

5.8 LLMs for Hardware Architecture Design

For hardware architecture design, LLMs have been explored in two primary areas: ***circuit architecture design*** [134, 226, 227] and ***specification document processing*** [225]. The comparison and timeline of these works are shown in Table 12. These works aim to leverage LLMs to enhance automation, reduce design complexity, and improve efficiency in architectural decision-making.

In ***circuit architecture design***, GPT4AIGChip [134] proposes an automated prompt-generation pipeline using in-context learning to guide LLMs in generating high-quality AI accelerator designs. This approach enables the structured decomposition of hardware design tasks, improving the consistency and efficiency of generated architectures. LCDA [226] applies LLMs to accelerate the software-hardware co-design process, particularly for compute-in-memory architectures in AI accelerators. It addresses the cold-start problem in traditional co-design approaches by leveraging LLMs to guide design space exploration, significantly reducing the search time. QGAS [227] extends the application of LLMs to quantum computing, using GPT-4 to iteratively design variational quantum algorithm ansatz architectures and translate the architecture into quantum assembly language code.

For ***specification document processing***, SpecLLM [225] tackles the inefficiencies and error-prone nature of developing architecture specifications in architecture design. It explores the use of LLMs to automate both the generation of specifications and the review of existing documentation. To structure the problem, the authors categorize architecture specifications into three levels, covering different degrees of design abstraction. They also introduce a dataset of 46 documents to evaluate the effectiveness of their approach. By leveraging LLMs, SpecLLM enhances both efficiency and accuracy in specification drafting and validation, demonstrating the potential for further automation in this critical aspect of hardware design.

5.9 LLMs for Analog Circuit Design.

While most research on LLMs for hardware design has focused on *digital* VLSI circuits, recent studies have started exploring LLM's potential in *analog circuit design*. Unlike digital design, which follows well-defined logic rules, analog circuits typically require tuning and optimization based on human

expertise, making LLM-assisted automation more challenging. A summary of existing works is provided in Table 13. These studies introduced LLM-powered frameworks targeting different analog circuit types, such as power converters and amplifiers, focusing on knowledge-based reasoning, topology synthesis, and circuit optimization. These approaches aim to enhance the efficiency of analog design, addressing its inherent complexities.

LADAC [229] introduces an LLM-driven decision-making agent for analog design, incorporating a knowledge library and interactive tools to assist in transistor sizing and simulation. AnalogCoder [230] employs a training-free LLM approach for Python-based circuit generation, integrating prompt engineering and feedback mechanisms to refine designs. ADO-LLM [232] combines Bayesian Optimization with LLMs to improve circuit design efficiency, leveraging Gaussian Process models for systematic design space exploration and in-context learning for guided optimization. LaMAGIC [233] fine-tunes LLMs for analog topology generation, particularly for power converters, developing structured input-output representations that enhance circuit synthesis accuracy. Artisan [234] focuses on operational amplifier (opamp) design, integrating topology selection and parameter tuning while employing Tree-of-Thought (ToT) and Chain-of-Thought (CoT) reasoning to improve structured decision-making. LEDRO [235] enhances analog circuit sizing by using LLMs to refine design search regions, improving the efficiency of existing optimization methods. AnalogXpert [236] streamlines analog circuit topology synthesis by incorporating a subcircuit library for design space reduction and using CoT prompting and iterative proofreading for better design accuracy. AnalogGenie [237] introduces a generative AI framework that utilizes a large dataset of over 3,000 analog circuit topologies. It employs a GPT-based model for sequential pin connection prediction, offering a scalable and flexible approach to analog circuit generation. These works collectively demonstrate the growing potential of LLMs in automating complex aspects of analog design, paving the way for more efficient and scalable circuit synthesis methodologies.

6 CHALLENGES, DISCUSSION, AND POTENTIAL DIRECTIONS

Despite the significant advancements in circuit foundation models, several challenges remain in terms of model performance, scalability, data availability, and the integration of predictive circuit encoders and generative circuit decoders. Moreover, these challenges are closely interrelated, often affecting and amplifying one another. We believe addressing these challenges is crucial to further enhancing the effectiveness and applicability of foundation AI models in EDA. In this section, we discuss our observed challenges and potential research directions to further improve the effectiveness and applicability of foundation AI models in EDA.

6.1 Challenge 1: Circuit Foundation Model Generalization and Scalability

The development of circuit foundation models presents several challenges regarding generalization, performance, and scalability. If we can address these challenges, circuit foundation models can effectively support a wider range of design tasks while maintaining computational efficiency.

Towards more generalized circuit embeddings from circuit encoders. One of the key challenges is designing circuit encoders that generate generalized embeddings capable of capturing both the semantic and structural intrinsic properties of circuits. These embeddings should effectively support largely different downstream tasks. For instance, design quality evaluation relies heavily on structural characteristics, while functional reasoning and verification require a deep understanding of circuit semantics. This requires innovations in ML model architectures, self-supervised learning techniques tailored for circuits, multimodal fusion strategies, and cross-design-stage alignment. The integration of graph-based encoders with text-based LLMs has shown promise in capturing both structural and semantic information, as seen in works like CircuitFusion [97], NetTAG [113],

and ProgSG [100]. However, further advancements are needed to enhance representation learning across different abstraction levels while ensuring alignment between circuit modalities.

Towards reducing hallucinations of circuit decoders. Decoder-based models, particularly those leveraging LLMs, are prone to generating hallucinated or syntactically incorrect HDL code. Unlike natural language, circuit descriptions have strict syntax and correctness constraints, requiring verification and refinement mechanisms to ensure reliability. Approaches such as reinforcement learning with human feedback (RLHF), constraint-aware decoding, and post-generation validation can help mitigate hallucinations and improve the practicality of decoder-based circuit models.

Towards more scalable circuit foundation models. Scalability remains a critical challenge, particularly for large-scale circuit designs. Current models struggle with handling industrial-scale designs due to the complexity and size of modern VLSI circuits. Divide-and-conquer strategies, such as hierarchical modeling, circuit partitioning [172], and subgraph-based processing [97], offer potential solutions to improve scalability. By segmenting circuits into smaller, more manageable sub-circuits and processing them independently, models can maintain computational efficiency without sacrificing accuracy. Techniques such as progressive training, adaptive resolution encoding, and distributed processing can further enhance the scalability of circuit foundation models, enabling their deployment in large-scale EDA workflows.

6.2 Challenge 2: Circuit Data Avaliability

The effectiveness of circuit foundation models heavily depends on access to large and diverse datasets for pre-training and fine-tuning. While efforts such as OpenABC-D [124], CircuitNet [245], and DeepCircuitX [246] have contributed by collecting open-source circuit designs, obtaining a sufficiently large and labeled dataset remains a challenge. Privacy concerns, proprietary design restrictions, and the high cost of generating high-quality annotated data further limit dataset availability. Overcoming these barriers may require advancements in synthetic dataset generation or novel circuit data augmentation techniques.

Towards generating synthetic circuit datasets. One emerging approach to overcoming data scarcity is the generation of unlimited synthetic circuit datasets, which can be created using graph-based or text-based methods, as explored in [24, 247, 248]. Graph-based approaches can generate large-scale circuit graphs with diverse topologies but often lack meaningful semantics, making it difficult to ensure functional correctness. On the other hand, text-based synthesis methods, such as automated HDL code generation, can produce realistic functional modules but typically lack scalability and diversity in structural variations. Bridging the gap between these two approaches by incorporating both functional correctness and large-scale diversity remains an open research challenge.

Towards more advanced circuit data augmentation. Data augmentation techniques have been widely explored in machine learning to improve model generalization. In the circuit domain, functionally equivalent transformations, such as logic optimization from the logic synthesis tools, have been used to create diverse training samples [97, 110, 113]. However, existing augmentation strategies primarily focus on structural transformations while maintaining functional equivalence. Future advancements could explore more sophisticated augmentation techniques, such as e-graph rewriting for RTL designs [249] and netlists [250] for broader design space exploration. These techniques can further enhance the robustness of circuit foundation models, ensuring they learn richer representations while preserving key design constraints such as timing, power, and area.

6.3 Challenge 3: Bridging the Gap Between Circuit Encoder and Decoder

While circuit encoders and decoders have been developed separately to support predictive and generative tasks, unifying these two paradigms presents an opportunity to create a more powerful

circuit foundation model. By integrating learned embeddings from circuit encoders into decoder-based generative models, and leveraging synthetic circuit generation from decoders to enhance circuit foundation model pre-training, the capabilities of both sides can be significantly improved.

Towards leveraging encoder embeddings for decoder generation. Current circuit decoders, often based on pre-trained LLMs, generate circuit contexts without explicitly considering circuit embeddings learned by encoders. By leveraging circuit encoders to generate structured, functionally meaningful embeddings, decoders can refine their generation process to ensure greater correctness and design feasibility. One potential approach is integrating decoder-based circuit text generation with graph embeddings learned from circuit encoders, allowing decoders to generate RTL, netlist, or layout designs that align with realistic circuit representations.

Towards leveraging decoder generated circuits for enhancing circuit foundation models. Generating synthetic circuits at different abstraction levels (e.g., RTL, netlist, layout) not only improves data availability but also provides a valuable resource for pre-training both encoders and decoders. By training foundation models on synthetically generated yet functionally diverse circuits, models can capture richer design patterns and structural relationships. Additionally, synthetic circuits can be used to fine-tune models for specific design tasks, enhancing their generalization across unseen circuit designs. Future research could explore hybrid approaches that combine rule-based generation, reinforcement learning, and generative models to create high-quality synthetic datasets that support both encoder and decoder training.

7 CONCLUSION

In this survey, we provide a systematic review of the latest progress in circuit foundation models, categorizing them into encoder-based and decoder-based approaches. Encoders aim to learn generalized circuit embeddings through self-supervised pre-training techniques, supporting predictive tasks such as design quality estimation and functional verification. Decoders, primarily built upon pre-trained LLMs, focus on generative tasks such as HDL code generation and verification automation. As AI techniques continue to transform the EDA landscape, circuit foundation models hold the potential to significantly reduce design effort, accelerate the chip design process, and improve design quality. Future potential research may target enhancing scalability, generalization, and efficiency, ultimately driving AI-powered innovation in modern VLSI design.

8 ACKNOWLEDGMENTS

This work is supported by Hong Kong Research Grants Council (RGC) CRF Grant C6003-24Y and ACCESS – AI Chip Center for Emerging Smart Systems, sponsored by InnoHK, Hong Kong SAR.

REFERENCES

[1] Ajay Tirumala and Raymond Wong. NVIDIA Blackwell Platform: Advancing Generative AI and Accelerated Computing. In *Hot Chips Symposium (HCS)*, 2024.
[2] IBS. As chip design costs skyrocket, 3nm process node is in jeopardy, 2020.
[3] Guyue Huang, Jingbo Hu, Yifan He, Jialong Liu, Mingyuan Ma, Zhaoyang Shen, Juejian Wu, Yuanfan Xu, Hengrui Zhang, Kai Zhong, et al. Machine learning for electronic design automation: A survey. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 2021.
[4] Martin Rapp, Hussam Amrouch, Yibo Lin, Bei Yu, David Z Pan, Marilyn Wolf, and Jörg Henkel. MLCAD: A survey of research in machine learning for CAD keynote paper. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 2021.
[5] Synopsys. DSO.ai: AI-driven design applications, 2021.
[6] Cadence. Cadence Cerebrus intelligent chip explorer, 2021.
[7] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

[8] Alec Radford, Jong Wook Kim, Chris Hallacy, Aditya Ramesh, Gabriel Goh, Sandhini Agarwal, Girish Sastry, Amanda Askell, Pamela Mishkin, Jack Clark, et al. Learning transferable visual models from natural language supervision. In *International Conference on Machine Learning (ICML)*, 2021.

[9] Aditya Ramesh, Prafulla Dhariwal, Alex Nichol, Casey Chu, and Mark Chen. Hierarchical text-conditional image generation with clip latents. *arXiv preprint arXiv:2204.06125*, 2022.

[10] Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, et al. GPT-4 technical report. *arXiv preprint arXiv:2303.08774*, 2023.

[11] Lei Chen, Yiqi Chen, Zhufei Chu, Wenji Fang, Tsung-Yi Ho, Ru Huang, Yu Huang, Sadaf Khan, Min Li, Xingquan Li, et al. Large circuit models: opportunities and challenges. *Springer Science China Information Sciences (SCIS)*, 2024.

[12] Yao Lu, Shang Liu, Qijun Zhang, and Zhiyao Xie. RTLLM: An open-source benchmark for design rtl generation with large language model. In *Asia and South Pacific Design Automation Conference (ASP-DAC)*, 2024.

[13] Mingjie Liu, Teodor-Dumitru Ene, Robert Kirby, Chris Cheng, Nathaniel Pinckney, Rongjian Liang, Jonah Alben, Himyanshu Anand, Sanmitra Banerjee, Ismet Bayraktaroglu, et al. ChipNeMo: Domain-Adapted LLMs for Chip Design. *arXiv preprint arXiv:2311.00176*, 2023.

[14] Mingjie Liu, Nathaniel Pinckney, Brucek Khailany, and Haoxing Ren. Verilogeval: Evaluating large language models for verilog code generation. *arXiv preprint arXiv:2309.07544*, 2023.

[15] Shailja Thakur, Baleegh Ahmad, Zhenxing Fan, Hammond Pearce, Benjamin Tan, Ramesh Karri, Brendan Dolan-Gavitt, and Siddharth Garg. Benchmarking large language models for automated verilog rtl code generation. In *Design, Automation and Test in Europe Conference and Exhibition (DATE)*, 2023.

[16] Haoxing Ren and Jiang Hu. *Machine Learning Applications in Electronic Design Automation*. Springer, 2022.

[17] Wenji Fang, Shang Liu, Hongce Zhang, and Zhiyao Xie. Annotating slack directly on your verilog: Fine-grained rtl timing evaluation for early optimization. In *Design Automation Conference (DAC)*, 2024.

[18] Ziyi Wang, Siting Liu, Yuan Pu, Song Chen, Tsung-Yi Ho, and Bei Yu. Restructure-tolerant timing prediction via multimodal fusion. In *Design Automation Conference (DAC)*, 2023.

[19] Zizheng Guo, Mingjie Liu, Jiaqi Gu, Shuhan Zhang, David Z Pan, and Yibo Lin. A timing engine inspired graph neural network model for pre-routing slack prediction. In *Design Automation Conference (DAC)*, 2022.

[20] Erick Carvajal Barboza, Nishchal Shukla, Yiran Chen, and Jiang Hu. Machine learning-based pre-routing timing prediction with reduced pessimism. In *Design Automation Conference (DAC)*, 2019.

[21] Andrew B Kahng, Uday Mallappa, and Lawrence Saul. Using machine learning to predict path-based slack from graph-based timing analysis. In *International Conference on Computer Design (ICCD)*, 2018.

[22] Zhiyao Xie, Rongjian Liang, Xiaoqing Xu, Jiang Hu, Chen-Chia Chang, Jingyu Pan, and Yiran Chen. Pre-placement net length and timing estimation by customized graph neural network. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 2022.

[23] Wenji Fang, Yao Lu, Shang Liu, Qijun Zhang, Ceyu Xu, Lisa Wu Wills, Hongce Zhang, and Zhiyao Xie. MasterRTL: A pre-synthesis PPA estimation framework for any RTL design. In *International Conference on Computer-Aided Design (ICCAD)*, 2023.

[24] Wenji Fang, Yao Lu, Shang Liu, Qijun Zhang, Ceyu Xu, Lisa Wu Wills, Hongce Zhang, and Zhiyao Xie. Transferable pre-synthesis PPA estimation for RTL designs with data augmentation techniques. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 2024.

[25] Ceyu Xu, Pragya Sharma, Tianshu Wang, and Lisa Wu Wills. Fast, robust and transferable prediction for hardware logic synthesis. In *International Symposium on Microarchitecture (MICRO)*, 2023.

[26] Prianka Sengupta, Aakash Tyagi, Yiran Chen, and Jiang Hu. How good is your Verilog RTL code? A quick answer from machine learning. In *International Conference on Computer-Aided Design (ICCAD)*, 2022.

[27] Qijun Zhang, Yao Lu, Mengming Li, and Zhiyao Xie. Autopower: Automated few-shot architecture-level power modeling by power group decoupling. In *Design Automation Conference (DAC)*, 2025.

[28] Qijun Zhang, Mengming Li, Yao Lu, and Zhiyao Xie. Firepower: Towards a foundation with generalizable knowledge for architecture-level power modeling. In *Asia and South Pacific Design Automation Conference (ASP-DAC)*, 2025.

[29] Yufan Du, Zizheng Guo, Xun Jiang, Zhuomin Chai, Yuxiang Zhao, Yibo Lin, Runsheng Wang, and Ru Huang. Powpredict: Cross-stage power prediction with circuit-transformation-aware learning. In *Design Automation Conference (DAC)*, 2024.

[30] Qijun Zhang, Shiyu Li, Guanglei Zhou, Jingyu Pan, Chen-Chia Chang, Yiran Chen, and Zhiyao Xie. Panda: Architecture-level power evaluation by unifying analytical and machine learning solutions. In *International Conference on Computer Aided Design (ICCAD)*, 2023.

[31] Zhiyao Xie, Xiaoqing Xu, Matt Walker, Joshua Knebel, Kumaraguru Palaniswamy, Nicolas Hebert, Jiang Hu, Huanrui Yang, Yiran Chen, and Shidhartha Das. APOLLO: An automated power modeling framework for runtime power introspection in high-volume commercial microprocessors. In *International Symposium on Microarchitecture (MICRO)*,

2021.

[32] Yuan Zhou, Haoxing Ren, Yanqing Zhang, Ben Keller, Brucek Khailany, and Zhiru Zhang. PRIMAL: Power inference using machine learning. In *Design Automation Conference (DAC)*, 2019.

[33] Donggyu Kim, Jerry Zhao, Jonathan Bachrach, and Krste Asanović. Simmani: Runtime power modeling for arbitrary RTL with automatic signal selection. In *International Symposium on Microarchitecture (MICRO)*, 2019.

[34] Zhiyao Xie, Shiyu Li, Mingyuan Ma, Chen-Chia Chang, Jingyu Pan, Yiran Chen, and Jiang Hu. DEEP: Developing extremely efficient runtime on-chip power meters. In *International Conference on Computer-Aided Design (ICCAD)*, 2022.

[35] Yanqing Zhang, Haoxing Ren, and Brucek Khailany. GRANNITE: Graph neural network inference for transferable power estimation. In *Design Automation Conference (DAC)*, 2020.

[36] Zhiyao Xie, Haoxing Ren, Brucek Khailany, Ye Sheng, Santosh Santosh, Jiang Hu, and Yiran Chen. PowerNet: Transferable dynamic IR drop estimation via maximum convolutional neural network. In *Asia and South Pacific Design Automation Conference (ASP-DAC)*, 2020.

[37] Chia-Tung Ho and Andrew B Kahng. IncPIRD: Fast learning-based prediction of incremental IR drop. In *International Conference on Computer-Aided Design (ICCAD)*, 2019.

[38] Zhiyao Xie, Haoxing Ren, Brucek Khailany, and Sheng Ye. IR drop prediction with maximum convolutional neural network, 2023. US Patent 11,645,533.

[39] Vidya A Chhabria, Yanqing Zhang, Haoxing Ren, Ben Keller, Brucek Khailany, and Sachin S Sapatnekar. MAVIREC: Ml-aided vectored ir-drop estimation and classification. In *Design, Automation and Test in Europe Conference and Exhibition (DATE)*, 2021.

[40] Yen-Chun Fang, Heng-Yi Lin, Min-Yan Sui, Chien-Mo Li, and Eric Jia-Wei Fang. Machine-learning-based dynamic IR drop prediction for ECO. In *International Conference on Computer-Aided Design (ICCAD)*, 2018.

[41] Zhiyao Xie, Yu-Hung Huang, Guan-Qi Fang, Haoxing Ren, Shao-Yun Fang, Yiran Chen, and Jiang Hu. RouteNet: Routability prediction for mixed-size designs using convolutional neural network. In *International Conference on Computer-Aided Design (ICCAD)*, 2018.

[42] Siting Liu, Qi Sun, Peiyu Liao, Yibo Lin, and Bei Yu. Global placement with deep learning-enabled explicit routability optimization. In *Design, Automation and Test in Europe Conference and Exhibition (DATE)*, 2021.

[43] Su Zheng, Lancheng Zou, Peng Xu, Siting Liu, Bei Yu, and Martin Wong. Lay-net: Grafting netlist knowledge on layout-based congestion prediction. In *International Conference on Computer-Aided Design (ICCAD)*, 2023.

[44] Jingsong Chen, Jian Kuang, Guowei Zhao, Dennis J-H Huang, and Evangeline FY Young. PROS: A plug-in for routability optimization applied in the state-of-the-art commercial EDA tool using deep learning. In *International Conference on Computer-Aided Design (ICCAD)*, 2020.

[45] Chen-Chia Chang, Jingyu Pan, Tunhou Zhang, Zhiyao Xie, Jiang Hu, Weiyi Qi, Chunwei Lin, Rongjian Liang, Joydeep Mitra, Elias Fallon, and Yiran Chen. Automatic routability predictor development using neural architecture search. In *International Conference on Computer-Aided Design (ICCAD)*, 2021.

[46] Jingyu Pan, Chen-Chia Chang, Zhiyao Xie, Ang Li, Minxue Tang, Tunhou Zhang, Jiang Hu, and Yiran Chen. Towards collaborative intelligence: Routability estimation based on decentralized private data. In *Design Automation Conference (DAC)*, 2022.

[47] Yu-Hung Huang, Zhiyao Xie, Guan-Qi Fang, Tao-Chun Yu, Haoxing Ren, Shao-Yun Fang, Yiran Chen, and Jiang Hu. Routability-driven macro placement with embedded cnn-based prediction model. In *Design, Automation and Test in Europe Conference and Exhibition (DATE)*, 2019.

[48] Rongjian Liang, Zhiyao Xie, Jinwook Jung, Vishnavi Chauha, Yiran Chen, Jiang Hu, Hua Xiang, and Gi-Joon Nam. Routing-free crosstalk prediction. In *International Conference on Computer-Aided Design (ICCAD)*, 2020.

[49] Martin Kuhlmann and Sachin S Sapatnekar. Exact and efficient crosstalk estimation. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 2001.

[50] Andrew B Kahng, Mulong Luo, and Siddhartha Nath. Si for free: machine learning of interconnect coupling delay and transition effects. In *International Workshop on System Level Interconnect Prediction (SLIP)*, 2015.

[51] Haoyu Yang, Jing Su, Yi Zou, Bei Yu, and Evangeline FY Young. Layout hotspot detection with feature tensor generation and deep biased learning. In *Design Automation Conference (DAC)*, 2017.

[52] Hao Geng, Haoyu Yang, Lu Zhang, Jin Miao, Fan Yang, Xuan Zeng, and Bei Yu. Hotspot detection via attention-based deep layout metric learning. In *International Conference on Computer-Aided Design (ICCAD)*, 2020.

[53] Haoyu Yang, Yajun Lin, Bei Yu, and Evangeline FY Young. Lithography hotspot detection: From shallow to deep learning. In *International System-on-Chip Conference (SOCC)*, 2017.

[54] Nan Wu, Yingjie Li, Cong Hao, Steve Dai, Cunxi Yu, and Yuan Xie. Gamora: Graph learning based symbolic reasoning for large-scale boolean networks. In *Design Automation Conference (DAC)*, 2023.

[55] Lilas Alrahis, Abhrajit Sengupta, Johann Knechtel, Satwik Patnaik, Hani Saleh, Baker Mohammad, Mahmoud Al-Qutayri, and Ozgur Sinanoglu. Gnn-re: Graph neural networks for reverse engineering of gate-level netlists. *IEEE*

Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD), 2021.

[56] Subhajit Dutta Chowdhury, Kaixin Yang, and Pierluigi Nuzzo. Reignn: State register identification using graph neural networks for circuit reverse engineering. In *International Conference on Computer-Aided Design (ICCAD)*, 2021.

[57] Zhuolun He, Ziyi Wang, Chen Bai, Haoyu Yang, and Bei Yu. Graph learning-based arithmetic block identification. In *International Conference on Computer-Aided Design (ICCAD)*, 2021.

[58] Yuzhe Ma, Haoxing Ren, Brucek Khailany, Harbinder Sikka, Lijuan Luo, Karthikeyan Natarajan, and Bei Yu. High performance graph convolutional networks with applications in testability analysis. In *Design Automation Conference (DAC)*, 2019.

[59] Zhiyao Xie, Guan-Qi Fang, Yu-Hung Huang, Haoxing Ren, Yanqing Zhang, Brucek Khailany, Shao-Yun Fang, Jiang Hu, Yiran Chen, and Erick Carvajal Barboza. FIST: A feature-importance sampling and tree-based method for automatic design flow parameter tuning. In *Asia and South Pacific Design Automation Conference (ASP-DAC)*, 2020.

[60] Walter Lau Neto, Yingjie Li, Pierre-Emmanuel Gaillardon, and Cunxi Yu. Flowtune: End-to-end automatic logic optimization exploration via domain-specific multiarmed bandit. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 2022.

[61] Rongjian Liang, Jinwook Jung, Hua Xiang, Lakshmi Reddy, Alexey Lvov, Jiang Hu, and Gi-Joon Nam. Flowtuner: A multi-stage eda flow tuner exploiting parameter knowledge transfer. In *International Conference on Computer-Aided Design (ICCAD)*, 2021.

[62] Chen Bai, Qi Sun, Jianwang Zhai, Yuzhe Ma, Bei Yu, and Martin DF Wong. Boom-explorer: Risc-v boom microarchitecture design space exploration framework. In *International Conference on Computer-Aided Design (ICCAD)*, 2021.

[63] Hung-Yi Liu and Luca P Carloni. On learning-based methods for design-space exploration with high-level synthesis. In *Design Automation Conference (DAC)*, 2013.

[64] Benjamin Carrion Schafer and Zi Wang. High-level synthesis design space exploration: Past, present, and future. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 2019.

[65] Yi-Chen Lu, Siddhartha Nath, Vishal Khandelwal, and Sung Kyu Lim. RL-sizer: VLSI gate sizing for timing optimization using deep reinforcement learning. In *Design Automation Conference (DAC)*, 2021.

[66] Yi-Chen Lu, Wei-Ting Chan, Deyuan Guo, Sudipto Kundu, Vishal Khandelwal, and Sung Kyu Lim. Rl-ccd: Concurrent clock and data optimization using attention-based self-supervised reinforcement learning. In *Design Automation Conference (DAC)*, 2023.

[67] Ruizhe Zhong, Xingbo Du, Shixiong Kai, Zhentao Tang, Siyuan Xu, Hui-Ling Zhen, Jianye Hao, Qiang Xu, Mingxuan Yuan, and Junchi Yan. Llm4eda: Emerging progress in large language models for electronic design automation. *arXiv preprint arXiv:2401.12224*, 2023.

[68] Jingyu Pan, Guanglei Zhou, Chen-Chia Chang, Isaac Jacobson, Jiang Hu, and Yiran Chen. A survey of research in large language models for electronic design automation. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 2025.

[69] Meisam Abdollahi, Seyedeh Faegheh Yeganli, Mohammad Amir Baharloo, and Amirali Baniasadi. Hardware design and verification with large language models: A scoping review, challenges, and open issues. *MDPI Electronics*, 2025.

[70] Dipayan Saha, Shams Tarek, Katayoon Yahyaei, Sujan Kumar Saha, Jingbo Zhou, Mark Tehranipoor, and Farimah Farahmandi. Llm for soc security: A paradigm shift. *IEEE Access*, 2024.

[71] Rahul Kande, Vasudev Gohil, Matthew DeLorenzo, Chen Chen, and Jeyavijayan Rajendran. Llms for hardware security: Boon or bane? In *VLSI Test Symposium (VTS)*, 2024.

[72] Zeng Wang, Lilas Alrahis, Likhitha Mankali, Johann Knechtel, and Ozgur Sinanoglu. Llms and the future of chip design: Unveiling security risks and building trust. In *Computer Society Annual Symposium on VLSI (ISVLSI)*, 2024.

[73] Sudipta Paria, Aritra Dasgupta, and Swarup Bhunia. Navigating soc security landscape on llm-guided paths. In *Great Lakes Symposium on VLSI (GLSVLSI)*, 2024.

[74] Kangwei Xu, Ruidi Qiu, Zhuorui Zhao, Grace Li Zhang, Ulf Schlichtmann, and Bing Li. Llm-aided efficient hardware design automation. *arXiv preprint arXiv:2410.18582*, 2024.

[75] Dheevatsa Mudigere, Yuchen Hao, Jianyu Huang, Zhihao Jia, Andrew Tulloch, Srinivas Sridharan, Xing Liu, Mustafa Ozdal, Jade Nie, Jongsoo Park, et al. Software-hardware co-design for fast and scalable training of deep learning recommendation models. In *International Symposium on Computer Architecture (ISCA)*, 2022.

[76] Chengtao Lai, Zhongchun Zhou, Akash Poptani, and Wei Zhang. Lcm: Llm-focused hybrid spm-cache architecture with cache management for multi-core ai accelerators. In *International Conference on Supercomputing (ICS)*, 2024.

[77] Guseul Heo, Sangyeop Lee, Jaehong Cho, Hyunmin Choi, Sanghyeon Lee, Hyungkyu Ham, Gwangsun Kim, Divya Mahajan, and Jongse Park. Neupims: Npu-pim heterogeneous acceleration for batched llm inferencing. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2024.

[78] Robert Brayton and Alan Mishchenko. Abc: An academic industrial-strength verification tool. In *International Conference on Computer Aided Verification (CAV)*, 2010.

[79] Gary D Hachtel and Fabio Somenzi. *Logic synthesis and verification algorithms*. Springer Science & Business Media, 2005.

[80] Bruce W. Ballard, John C. Lusth, and Nancy L. Tinkham. Ldc-1: a transportable, knowledge-based natural language processor for office environments. *ACM Transactions on Information Systems (TOIS)*, 1984.

[81] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781*, 2013.

[82] Jeffrey Pennington, Richard Socher, and Christopher Manning. GloVe: Global vectors for word representation. In *Conference on Empirical Methods in Natural Language Processing (EMNLP)*, 2014.

[83] Tomas Mikolov, Martin Karafiat, Lukavs Burget, Jan Honza Cernocky, and Sanjeev Khudanpur. Recurrent neural network based language model. In *Interspeech*, 2010.

[84] H Sak. Long short-term memory based recurrent neural network architectures for large vocabulary speech recognition. *arXiv preprint arXiv:1402.1128*, 2014.

[85] Ting Chen, Simon Kornblith, Mohammad Norouzi, and Geoffrey Hinton. A simple framework for contrastive learning of visual representations. In *International Conference on Machine Learning (ICML)*, 2020.

[86] Kaiming He, Haoqi Fan, Yuxin Wu, Saining Xie, and Ross Girshick. Momentum contrast for unsupervised visual representation learning. In *Conference on Computer Vision and Pattern Recognition (CVPR)*, 2020.

[87] Kaiming He, Xinlei Chen, Saining Xie, Yanghao Li, Piotr Dollár, and Ross Girshick. Masked autoencoders are scalable vision learners. In *Conference on Computer Vision and Pattern Recognition (CVPR)*, 2022.

[88] Long Ouyang, Jeff Wu, Xu Jiang, Diogo Almeida, Carroll L. Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, John Schulman, Jacob Hilton, Fraser Kelton, Luke Miller, Maddie Simens, Amanda Askell, Peter Welinder, Paul Christiano, Jan Leike, and Ryan Lowe. Training language models to follow instructions with human feedback. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2024.

[89] Aaron Van den Oord, Nal Kalchbrenner, Lasse Espeholt, Oriol Vinyals, Alex Graves, and Koray Kavukcuoglu. Conditional image generation with pixelcnn decoders. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2016.

[90] Junnan Li, Ramprasaath Selvaraju, Akhilesh Gotmare, Shafiq Joty, Caiming Xiong, and Steven Chu Hong Hoi. Align before fuse: Vision and language representation learning with momentum distillation. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2021.

[91] Junnan Li, Dongxu Li, Caiming Xiong, and Steven Hoi. Blip: Bootstrapping language-image pre-training for unified vision-language understanding and generation. In *International Conference on Machine Learning (ICML)*, 2022.

[92] Junnan Li, Dongxu Li, Silvio Savarese, and Steven Hoi. Blip-2: Bootstrapping language-image pre-training with frozen image encoders and large language models. In *International Conference on Machine Learning (ICML)*, 2023.

[93] Haotian Liu, Chunyuan Li, Qingyang Wu, and Yong Jae Lee. Visual instruction tuning. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2024.

[94] Bin Lin, Yang Ye, Bin Zhu, Jiaxi Cui, Munan Ning, Peng Jin, and Li Yuan. Video-llava: Learning united visual representation by alignment before projection. *arXiv preprint arXiv:2311.10122*, 2023.

[95] Aditya Ramesh, Mikhail Pavlov, Gabriel Goh, Scott Gray, Chelsea Voss, Alec Radford, Mark Chen, and Ilya Sutskever. Zero-shot text-to-image generation. In *International Conference on Machine Learning (ICML)*, 2021.

[96] Jiahui Yu, Yuanzhong Xu, Jing Yu Koh, Thang Luong, Gunjan Baid, Zirui Wang, Vijay Vasudevan, Alexander Ku, Yinfei Yang, Burcu Karagol Ayan, et al. Scaling autoregressive models for content-rich text-to-image generation. *arXiv preprint arXiv:2206.10789*, 2022.

[97] Wenji Fang, Shang Liu, Jing Wang, and Zhiyao Xie. Circuitfusion: multimodal circuit representation learning for agile chip design. In *International Conference on Learning Representations (ICLR)*, 2025.

[98] Zhengyuan Shi, Ziyang Zheng, Sadaf Khan, Jianyuan Zhong, Min Li, and Qiang Xu. Deepgate3: towards scalable circuit representation learning. *arXiv preprint arXiv:2407.11095*, 2024.

[99] Atefeh Sohrabizadeh, Yunsheng Bai, Yizhou Sun, and Jason Cong. Robust gnn-based representation learning for hls. In *International Conference on Computer-Aided Design (ICCAD)*, 2023.

[100] Zongyue Qin, Yunsheng Bai, Atefeh Sohrabizadeh, Zijian Ding, Yizhou Sun, and Jason Cong. Cross-modality program representation learning for electronic design automation with high-level synthesis. In *International Symposium on Machine Learning for CAD (MLCAD)*, 2024.

[101] Shobha Vasudevan, Wenjie Joe Jiang, David Bieber, Rishabh Singh, C Richard Ho, Charles Sutton, et al. Learning semantic representations to verify hardware designs. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2021.

[102] Wenji Fang, Shang Liu, Hongce Zhang, and Zhiyao Xie. A self-supervised, pre-trained, and cross-stage-aligned circuit encoder provides a foundation for various design tasks. In *Asia and South Pacific Design Automation Conference (ASP-DAC)*, 2025.

[103] Min Li, Sadaf Khan, Zhengyuan Shi, Naixing Wang, Huang Yu, and Qiang Xu. DeepGate: Learning neural representations of logic gates. In *Design Automation Conference (DAC)*, 2022.

[104] Zhengyuan Shi, Hongyang Pan, Sadaf Khan, Min Li, Yi Liu, Junhua Huang, Hui-Ling Zhen, Mingxuan Yuan, Zhufei Chu, and Qiang Xu. DeepGate2: Functionality-aware circuit representation learning. In *International Conference on Computer-Aided Design (ICCAD)*, 2023.

[105] Ziyang Zheng, Shan Huang, Jianyuan Zhong, Zhengyuan Shi, Guohao Dai, Ningyi Xu, and Qiang Xu. Deepgate4: Efficient and effective representation learning for circuit design at scale. In *International Conference on Learning Representations (ICLR)*, 2025.

[106] Chenhui Deng, Zichao Yue, Cunxi Yu, Gokce Sarar, Ryan Carey, Rajeev Jain, and Zhiru Zhang. Less is more: Hop-wise graph attention for scalable and generalizable learning on circuits. In *Design Automation Conference (DAC)*, 2024.

[107] Jiawei Liu, Jianwang Zhai, Mingyu Zhao, Zhe Lin, Bei Yu, and Chuan Shi. Polargate: Breaking the functionality representation bottleneck of and-inverter graph neural network. In *International Conference on Computer-Aided Design (ICCAD)*, 2024.

[108] Sadaf Khan, Zhengyuan Shi, Min Li, and Qiang Xu. Deepseq: Deep sequential circuit learning. In *Design, Automation and Test in Europe Conference and Exhibition (DATE)*, 2024.

[109] Sadaf Khan, Zhengyuan Shi, Ziyang Zheng, Min Li, and Qiang Xu. Deepseq2: Enhanced sequential circuit learning with disentangled representations. In *Asia and South Pacific Design Automation Conference (ASP-DAC)*, 2025.

[110] Ziyi Wang, Chen Bai, Zhuolun He, Guangliang Zhang, Qiang Xu, Tsung-Yi Ho, Bei Yu, and Yu Huang. Functionality matters in netlist representation learning. In *Design Automation Conference (DAC)*, 2022.

[111] Ziyi Wang, Chen Bai, Zhuolun He, Guangliang Zhang, Qiang Xu, Tsung-Yi Ho, Yu Huang, and Bei Yu. Fgnn2: A powerful pre-training framework for learning the logic functionality of circuits. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 2024.

[112] Haoyuan Wu, Haisheng Zheng, Yuan Pu, and Bei Yu. Circuit representation learning with masked gatemodeling and verilog-aigalignment. In *International Conference on Learning Representations (ICLR)*, 2025.

[113] Wenji Fang, Wenkai Li, Shang Liu, Yao Lu, Hongce Zhang, and Zhiyao Xie. Nettag: A multimodal rtl-and-layout-aligned netlist foundation model via text-attributed graph. In *Design Automation Conference (DAC)*, 2025.

[114] Zhengyuan Shi, Chengyu Ma, Ziyang Zheng, Lingfeng Zhou, Hongyang Pan, Wentao Jiang, Fan Yang, Xiaoyan Yang, Zhufei Chu, and Qiang Xu. Deepcell: Multiview representation learning for post-mapping netlists. *arXiv preprint arXiv:2502.06816*, 2025.

[115] Shuwen Yang, Zhihao Yang, Dong Li, Yingxueff Zhang, Zhanguang Zhang, Guojie Song, and Jianye Hao. Versatile multi-stage graph neural network for circuit representation. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2022.

[116] Keren Zhu, Hao Chen, Walker J Turner, George F Kokai, Po-Hsuan Wei, David Z Pan, and Haoxing Ren. Tag: Learning circuit spatial embedding from layouts. In *International Conference on Computer-Aided Design (ICCAD)*, 2022.

[117] Yuyang Chen, Yiwen Wu, Jingya Wang, Tao Wu, Xuming He, Jingyi Yu, and Hao Geng. Llm-hd: Layout language model for hotspot detection with gds semantic encoding. In *Design Automation Conference (DAC)*, 2024.

[118] Yunsheng Bai, Atefeh Sohrabizadeh, Zongyue Qin, Ziniu Hu, Yizhou Sun, and Jason Cong. Towards a comprehensive benchmark for high-level synthesis targeted to fpgas. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2023.

[119] Fulvio Corno, Matteo Sonza Reorda, and Giovanni Squillero. RT-level ITC'99 benchmarks and first ATPG results. *IEEE Design and Test of Computers*, 2000.

[120] *OpenCores: The reference community for Free and Open Source gateware IP cores.*

[121] Alon Amid, David Biancolin, Abraham Gonzalez, Daniel Grubb, Sagar Karandikar, Harrison Liew, Albert Magyar, Howard Mao, Albert Ou, Nathan Pemberton, et al. Chipyard: Integrated design, simulation, and implementation framework for custom SoCs. *IEEE Micro*, 2020.

[122] VexRiscv. VexRiscv: A FPGA friendly 32 bit RISC-V CPU implementation, 2022.

[123] Yinan Xu, Zihao Yu, Dan Tang, Guokai Chen, Lu Chen, Lingrui Gou, Yue Jin, Qianruo Li, Xin Li, Zuojun Li, et al. Towards developing high performance risc-v processors using agile methodology. In *International Symposium on Microarchitecture (MICRO)*, 2022.

[124] Animesh Basak Chowdhury, Benjamin Tan, Ramesh Karri, and Siddharth Garg. Openabc-d: A large-scale dataset for machine learning guided integrated circuit synthesis. *arXiv preprint arXiv:2110.11292*, 2021.

[125] Christoph Albrecht. IWLS 2005 benchmarks. In *International Workshop on Logic and Synthesis (IWLS)*, 2005.

[126] Luca Amarú, Pierre-Emmanuel Gaillardon, and Giovanni De Micheli. The epfl combinational benchmark suite. In *International Workshop on Logic and Synthesis (IWLS)*, 2015.

[127] Ken McElvain. Lgsynth93 benchmark set: Version 4.0. *Mentor Graphics, May*, 1993.

[128] Natarajan Viswanathan, Charles J Alpert, Cliff Sze, Zhuo Li, Gi-Joon Nam, and Jarrod A Roy. The ispd-2011 routability-driven placement contest and benchmark suite. In *International Symposium on Physical Design (ISPD)*,

2011.

[129] Natarajan Viswanathan, Charles Alpert, Cliff Sze, Zhuo Li, and Yaoguang Wei. The dac 2012 routability-driven placement contest and benchmark suite. In *Design Automation Conference (DAC)*, 2012.

[130] J Andres Torres. Iccad-2012 cad contest in fuzzy pattern matching for physical verification and benchmark suite. In *International Conference on Computer-Aided Design (ICCAD)*, 2012.

[131] Kai-Shun Hu, Ming-Jen Yang, Tao-Chun Yu, and Guan-Chuen Chen. Iccad-2020 cad contest in routing with cell movement. In *International Conference on Computer-Aided Design (ICCAD)*, 2020.

[132] Hammond Pearce, Benjamin Tan, and Ramesh Karri. Dave: Deriving automatically verilog from english. In *Workshop on Machine Learning for CAD (MLCAD)*, 2020.

[133] Kaiyan Chang, Ying Wang, Haimeng Ren, Mengdi Wang, Shengwen Liang, Yinhe Han, Huawei Li, and Xiaowei Li. Chipgpt: How far are we from natural language hardware design. *arXiv preprint arXiv:2305.14019*, 2023.

[134] Yonggan Fu, Yongan Zhang, Zhongzhi Yu, Sixu Li, Zhifan Ye, Chaojian Li, Cheng Wan, and Yingyan Celine Lin. GPT4AIGChip: Towards next-generation AI accelerator design automation via large language models. In *International Conference on Computer-Aided Design (ICCAD)*, 2023.

[135] Jason Blocklove, Siddharth Garg, Ramesh Karri, and Hammond Pearce. Chip-chat: Challenges and opportunities in conversational hardware design. *arXiv preprint arXiv:2305.13243*, 2023.

[136] Shailja Thakur, Jason Blocklove, Hammond Pearce, Benjamin Tan, Siddharth Garg, and Ramesh Karri. Autochip: Automating hdl generation using llm feedback. *arXiv preprint arXiv:2311.04887*, 2023.

[137] Shailja Thakur, Baleegh Ahmad, Hammond Pearce, Benjamin Tan, Brendan Dolan-Gavitt, Ramesh Karri, and Siddharth Garg. Verigen: A large language model for verilog code generation. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 2024.

[138] Mubashir ul Islam, Humza Sami, Pierre-Emmanuel Gaillardon, and Valerio Tenace. Eda-aware rtl generation with large language models. *arXiv preprint arXiv:2412.04485*, 2024.

[139] Yang Zhao, Di Huang, Chongxiao Li, Pengwei Jin, Ziyuan Nan, Tianyun Ma, Lei Qi, Yansong Pan, Zhenxing Zhang, Rui Zhang, et al. Codev: Empowering llms for verilog generation through multi-level summarization. *arXiv preprint arXiv:2407.10424*, 2024.

[140] Mingzhe Gao, Jieru Zhao, Zhe Lin, Wenchao Ding, Xiaofeng Hou, Yu Feng, Chao Li, and Minyi Guo. Autovcoder: A systematic framework for automated verilog code generation using llms. In *International Conference on Computer Design (ICCD)*, 2024.

[141] Zehua Pei, Hui-Ling Zhen, Mingxuan Yuan, Yu Huang, and Bei Yu. Betterv: Controlled verilog generation with discriminative guidance. *arXiv preprint arXiv:2402.03375*, 2024.

[142] Kaiyan Chang, Kun Wang, Nan Yang, Ying Wang, Dantong Jin, Wenlong Zhu, Zhirong Chen, Cangyuan Li, Hao Yan, Yunhao Zhou, et al. Data is all you need: Finetuning llms for chip design via an automated design-data augmentation framework. *arXiv preprint arXiv:2403.11202*, 2024.

[143] Fan Cui, Chenyang Yin, Kexing Zhou, Youwei Xiao, Guangyu Sun, Qiang Xu, Qipeng Guo, Demin Song, Dahua Lin, Xingcheng Zhang, et al. Origen: Enhancing rtl code generation with code-to-code augmentation and self-reflection. *arXiv preprint arXiv:2407.16237*, 2024.

[144] Chia-Tung Ho, Haoxing Ren, and Brucek Khailany. Verilogcoder: Autonomous verilog coding agents with graph-based planning and abstract syntax tree (ast)-based waveform tracing tool. *arXiv preprint arXiv:2408.08927*, 2024.

[145] Shang Liu, Wenji Fang, Yao Lu, Qijun Zhang, Hongce Zhang, and Zhiyao Xie. Rtlcoder: Fully open-source and efficient llm-assisted rtl code generation technique. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 2024.

[146] Yongan Zhang, Zhongzhi Yu, Yonggan Fu, Cheng Wan, and Yingyan Celine Lin. Mg-verilog: Multi-grained dataset towards enhanced llm-assisted verilog generation. *arXiv preprint arXiv:2407.01910*, 2024.

[147] Matthew DeLorenzo, Vasudev Gohil, and Jeyavijayan Rajendran. Creativeval: Evaluating creativity of llm-based hardware code generation. *arXiv preprint arXiv:2404.08806*, 2024.

[148] Prashanth Vijayaraghavan, Luyao Shi, Stefano Ambrogio, Charles Mackin, Apoorva Nitsure, David Beymer, and Ehsan Degan. Vhdl-eval: A framework for evaluating large language models in vhdl code generation. In *LLM Aided Design Workshop (LAD)*, 2024.

[149] Kaiyan Chang, Zhirong Chen, Yunhao Zhou, Wenlong Zhu, Haobo Xu, Cangyuan Li, Mengdi Wang, Shengwen Liang, Huawei Li, Yinhe Han, et al. Natural language is not enough: Benchmarking multi-modal generative ai for verilog generation. *arXiv preprint arXiv:2407.08473*, 2024.

[150] Kimia Tasnia and Sazadur Rahman. Opl4gpt: An application space exploration of optimal programming language for hardware design by llm. In *Asia and South Pacific Design Automation Conference (ASP-DAC)*, 2025.

[151] Bowei Wang, Qi Xiong, Zeqing Xiang, Lei Wang, and Renzhi Chen. Rtlsquad: Multi-agent based interpretable rtl design. *arXiv preprint arXiv:2501.05470*, 2025.

[152] Yujie Zhao, Hejia Zhang, Hanxian Huang, Zhongming Yu, and Jishen Zhao. Mage: A multi-agent engine for automated rtl code generation. *arXiv preprint arXiv:2412.07822*, 2024.
[153] Ahmed Allam and Mohamed Shalan. Rtl-repo: A benchmark for evaluating llms on large-scale rtl design projects. *arXiv preprint arXiv:2405.17378*, 2024.
[154] Shang Liu, Yao Lu, Wenji Fang, Mengming Li, and Zhiyao Xie. Openllm-rtl: Open dataset and benchmark for llm-aided design rtl generation. In *International Conference on Computer-Aided Design (ICCAD)*, 2024.
[155] Wenhao Sun, Bing Li, Grace Li Zhang, Xunzhao Yin, Cheng Zhuo, and Ulf Schlichtmann. Classification-based automatic hdl code generation using llms. *arXiv preprint arXiv:2407.18326*, 2024.
[156] Yi Liu, Changran Xu, Yunhao Zhou, Zeju Li, and Qiang Xu. Deeprtl: Bridging verilog understanding and generation with a unified representation model. *arXiv preprint arXiv:2502.15832*, 2025.
[157] Mingjie Liu, Yun-Da Tsai, Wenfei Zhou, and Haoxing Ren. Craftrtl: High-quality synthetic data generation for verilog code models with correct-by-construction non-textual representations and targeted code repair. *arXiv preprint arXiv:2409.12993*, 2024.
[158] Madhav Nair, Rajat Sadhukhan, and Debdeep Mukhopadhyay. Generating secure hardware using chatgpt resistant to cwes. *Cryptology ePrint Archive*, 2023.
[159] Andre Nakkab, Sai Qian Zhang, Ramesh Karri, and Siddharth Garg. Rome was not built in a single step: Hierarchical prompting for llm-based chip design. *arXiv preprint arXiv:2407.18276*, 2024.
[160] Prashanth Vijayaraghavan, Apoorva Nitsure, Charles Mackin, Luyao Shi, Stefano Ambrogio, Arvind Haran, Viresh Paruthi, Ali Elzein, Dan Coops, David Beymer, et al. Chain-of-descriptions: Improving code llms for vhdl code generation and summarization. In *International Symposium on Machine Learning for CAD (MLCAD)*, 2024.
[161] Selim Sandal and Ismail Akturk. Zero-shot rtl code generation with attention sink augmented large language models. *arXiv preprint arXiv:2401.08683*, 2024.
[162] Matthew DeLorenzo, Animesh Basak Chowdhury, Vasudev Gohil, Shailja Thakur, Ramesh Karri, Siddharth Garg, and Jeyavijayan Rajendran. Make every move count: Llm-based high-quality rtl code generation using mcts. *arXiv preprint arXiv:2402.03289*, 2024.
[163] Ning Wang, Bingkun Yao, Jie Zhou, Xi Wang, Zhe Jiang, and Nan Guan. Large language model for verilog generation with golden code feedback. *arXiv preprint arXiv:2407.18271*, 2024.
[164] Emil Goh, Maoyang Xiang, I Wey, and T Hui Teo. From english to asic: Hardware implementation with large language model. *arXiv preprint arXiv:2403.07039*, 2024.
[165] Nathaniel Pinckney, Christopher Batten, Mingjie Liu, Haoxing Ren, and Brucek Khailany. Revisiting verilogeval: Newer llms, in-context learning, and specification-to-rtl tasks. *arXiv preprint arXiv:2408.11053*, 2024.
[166] Zhigang Fang, Renzhi Chen, Zhijie Yang, Yang Guo, Huadong Dai, and Lei Wang. Lintllm: An open-source verilog linting framework based on large language models. *arXiv preprint arXiv:2502.10815*, 2025.
[167] Chenwei Xiong, Cheng Liu, Huawei Li, and Xiaowei Li. Hlspilot: Llm-based high-level synthesis. *arXiv preprint arXiv:2408.06810*, 2024.
[168] Luca Collini, Siddharth Garg, and Ramesh Karri. C2hlsc: Leveraging large language models to bridge the software-to-hardware design gap. *arXiv preprint arXiv:2412.00214*, 2024.
[169] Seyed Arash Sheikholeslam and Andre Ivanov. Synthai: A multi agent generative ai framework for automated modular hls design generation. *arXiv preprint arXiv:2405.16072*, 2024.
[170] Yuchao Liao, Tosiron Adegbija, and Roman Lysecky. Are llms any good for high-level synthesis? *arXiv preprint arXiv:2408.10428*, 2024.
[171] Jiahao Gai, Hao Chen, Zhican Wang, Hongyu Zhou, Wanru Zhao, Nicholas Lane, and Hongxiang Fan. Exploring code language models for automated hls-based hardware generation: Benchmark, infrastructure and analysis. In *Asia and South Pacific Design Automation Conference (ASP-DAC)*, 2025.
[172] Xufeng Yao, Yiwen Wang, Xing Li, Yingzhao Lian, Ran Chen, Lei Chen, Mingxuan Yuan, Hong Xu, and Bei Yu. Rtlrewriter: Methodologies for large models aided rtl code optimization. *arXiv preprint arXiv:2409.11414*, 2024.
[173] Pablo Antonio Martínez, Gregorio Bernabé, and José Manuel García. Code detection for hardware acceleration using large language models. *IEEE Access*, 2024.
[174] Haocheng Xu, Haotian Hu, and Sitao Huang. Optimizing high-level synthesis designs with retrieval-augmented large language models. In *LLM Aided Design Workshop (LAD)*, 2024.
[175] Kiran Thorat, Jiahui Zhao, Yaotian Liu, Hongwu Peng, Xi Xie, Bin Lei, Jeff Zhang, and Caiwen Ding. Advanced large language model (llm)-driven verilog development: Enhancing power, performance, and area optimization in code synthesis. *arXiv preprint arXiv:2312.01022*, 2023.
[176] YunDa Tsai, Mingjie Liu, and Haoxing Ren. Rtlfixer: Automatically fixing rtl syntax errors with large language models. *arXiv preprint arXiv:2311.16543*, 2023.
[177] Marcelo Orenes-Vera, Margaret Martonosi, and David Wentzlaff. Using LLMs to facilitate formal verification of RTL. *arXiv preprint arXiv:2309.09437*, 2023.

[178] Xingyu Meng, Amisha Srivastava, Ayush Arunachalam, Avik Ray, Pedro Henrique Silva, Rafail Psiakis, Yiorgos Makris, and Kanad Basu. Unlocking hardware security assurance: The potential of llms. *arXiv preprint arXiv:2308.11042*, 2023.
[179] Sudipta Paria, Aritra Dasgupta, and Swarup Bhunia. Divas: An llm-based end-to-end framework for soc security analysis and policy-based protection. *arXiv preprint arXiv:2308.06932*, 2023.
[180] Mohammad Akyash and Hadi Mardani Kamali. Simeval: Investigating the similarity obstacle in llm-based hardware code generation. In *Asia and South Pacific Design Automation Conference (ASP-DAC)*, 2025.
[181] Zhiyuan Yan, Wenji Fang, Mengming Li, Min Li, Zhiyuan Yan, Shang Liu, Zhiyao Xie, and Hongce Zhang. AssertLLM: Generating and evaluating hardware verification assertions from design specifications via multi-LLMs. In *Asia and South Pacific Design Automation Conference (ASP-DAC)*, 2025.
[182] Bhabesh Mali, Karthik Maddala, Sweeya Reddy, Vatsal Gupta, Chandan Karfa, and Ramesh Karri. Chiraag: Chatgpt informed rapid and automated assertion generation. *arXiv preprint arXiv:2402.00093*, 2024.
[183] Yuchen Hu, Junhao Ye, Ke Xu, Jialin Sun, Shiyue Zhang, Xinyao Jiao, Dingrong Pan, Jie Zhou, Ning Wang, Weiwei Shan, et al. Uvllm: An automated universal rtl verification framework using llms. *arXiv preprint arXiv:2411.16238*, 2024.
[184] Zixi Zhang, Greg Chadwick, Hugo McNally, Yiren Zhao, and Robert Mullins. Llm4dv: Using large language models for hardware test stimuli generation. *arXiv preprint arXiv:2310.04535*, 2023.
[185] Ruiyang Ma, Yuxin Yang, Ziqian Liu, Jiaxi Zhang, Min Li, Junhua Huang, and Guojie Luo. Verilogreader: Llm-aided hardware test generation. *arXiv preprint arXiv:2406.04373*, 2024.
[186] Minwoo Kang, Mingjie Liu, Ghaith Bany Hamad, Syed Suhaib, and Haoxing Ren. Fveval: Understanding language model capabilities in formal verification of digital hardware. *arXiv preprint arXiv:2410.23299*, 2024.
[187] Vaishnavi Pulavarthi, Deeksha Nandal, Soham Dan, and Debjit Pal. Assertionbench: A benchmark to evaluate large-language models for assertion generation. *arXiv preprint arXiv:2406.18627*, 2024.
[188] Rahul Kande, Hammond Pearce, Benjamin Tan, Brendan Dolan-Gavitt, Shailja Thakur, Ramesh Karri, and Jeyavijayan Rajendran. (security) assertions by large language models. *IEEE Transactions on Information Forensics and Security (TIFS)*, 2024.
[189] Chuyue Sun, Christopher Hahn, and Caroline Trippel. Towards improving verification productivity with circuit-aware translation of natural language to systemverilog assertions. In *International Workshop on Deep Learning-aided Verification (DAV)*, 2023.
[190] Mingjie Liu, Minwoo Kang, Ghaith Bany Hamad, Syed Suhaib, and Haoxing Ren. Domain-adapted llms for vlsi design and verification: A case study on formal verification. In *VLSI Test Symposium (VTS)*, 2024.
[191] Hanxian Huang, Zhenghan Lin, Zixuan Wang, Xin Chen, Ke Ding, and Jishen Zhao. Towards llm-powered verilog rtl assistant: Self-verification and self-correction. *arXiv preprint arXiv:2406.00115*, 2024.
[192] Rahul Kande, Hammond Pearce, Benjamin Tan, Brendan Dolan-Gavitt, Shailja Thakur, Ramesh Karri, and Jeyavijayan Rajendran. Llm-assisted generation of hardware assertions. *arXiv preprint arXiv:2306.14027*, 2023.
[193] Baleegh Ahmad, Shailja Thakur, Benjamin Tan, Ramesh Karri, and Hammond Pearce. Fixing hardware security bugs with large language models. *arXiv preprint arXiv:2302.01215*, 2023.
[194] Chao Xiao, Yifei Deng, Zhijie Yang, Renzhi Chen, Hong Wang, Jingyue Zhao, Huadong Dai, Lei Wang, Yuhua Tang, and Weixia Xu. Llm-based processor verification: A case study for neuronnorphic processor. In *Design, Automation and Test in Europe Conference and Exhibition (DATE)*, 2024.
[195] Jitendra Bhandari, Johann Knechtel, Ramesh Narayanaswamy, Siddharth Garg, and Ramesh Karri. Llm-aided testbench generation and bug detection for finite-state machines. *arXiv preprint arXiv:2406.17132*, 2024.
[196] Jason Blocklove, Siddharth Garg, Ramesh Karri, and Hammond Pearce. Evaluating llms for hardware design and test. *arXiv preprint arXiv:2405.02326*, 2024.
[197] Jie Zhou, Youshu Ji, Ning Wang, Yuchen Hu, Xinyao Jiao, Bingkun Yao, Xinwei Fang, Shuai Zhao, Nan Guan, and Zhe Jiang. Insights from rights and wrongs: A large language model for solving assertion failures in rtl design. *arXiv preprint arXiv:2503.04057*, 2025.
[198] Vaishnavi Pulavarthi, Deeksha Nandal, Soham Dan, and Debjit Pal. Are llms ready for practical adoption for assertion generation? *arXiv preprint arXiv:2502.20633*, 2025.
[199] Ke Xu, Jialin Sun, Yuchen Hu, Xinwei Fang, Weiwei Shan, Xi Wang, and Zhe Jiang. Meic: Re-thinking rtl debug automation using llms. *arXiv preprint arXiv:2405.06840*, 2024.
[200] Xufeng Yao, Haoyang Li, Tsz Ho Chan, Wenyi Xiao, Mingxuan Yuan, Yu Huang, Lei Chen, and Bei Yu. Hdldebugger: Streamlining hdl debugging with large language models. *arXiv preprint arXiv:2403.11671*, 2024.
[201] Lily Jiaxin Wan, Yingbing Huang, Yuhong Li, Hanchen Ye, Jinghua Wang, Xiaofan Zhang, and Deming Chen. Invited paper: Software/hardware co-design for llm and its application for design verification. In *Asia and South Pacific Design Automation Conference (ASP-DAC)*, 2024.
[202] Weimin Fu, Kaichen Yang, Raj Gautam Dutta, Xiaolong Guo, and Gang Qu. Llm4sechw: Leveraging domain-specific large language model for hardware debugging. In *Asian Hardware Oriented Security and Trust Symposium (AsianHOST)*,

2023.

[203] Baleegh Ahmad, Shailja Thakur, Benjamin Tan, Ramesh Karri, and Hammond Pearce. On hardware security bug code fixes by prompting large language models. *IEEE Transactions on Information Forensics and Security (TIFS)*, 2024.

[204] Khushboo Qayyum, Muhammad Hassan, Sallar Ahmadi-Pour, Chandan Kumar Jha, and Rolf Drechsler. From bugs to fixes: Hdl bug identification and patching using llms and rag. In *LLM Aided Design Workshop (LAD)*, 2024.

[205] Dipayan Saha, Katayoon Yahyaei, Sujan Kumar Saha, Mark Tehranipoor, and Farimah Farahmandi. Empowering hardware security with llm: The development of a vulnerable hardware database. In *International Symposium on Hardware Oriented Security and Trust (HOST)*, 2024.

[206] Mohammad Akyash and Hadi Mardani Kamali. Self-hwdebug: Automation of llm self-instructing for hardware security verification. *arXiv preprint arXiv:2405.12347*, 2024.

[207] Baleegh Ahmad, Shailja Thakur, Benjamin Tan, Ramesh Karri, and Hammond Pearce. On hardware security bug code fixes by prompting large language models. *IEEE Transactions on Information Forensics and Security (TIFS)*, 2024.

[208] Marcelo Orenes-Vera, Margaret Martonosi, and David Wentzlaff. From rtl to sva: Llm-assisted generation of formal verification testbenches. *arXiv preprint arXiv:2309.09437*, 2023.

[209] Banafsheh Saber Latibari, Sujan Ghimire, Muhtasim Alam Chowdhury, Najmeh Nazari, Kevin Immanuel Gubbi, Houman Homayoun, Avesta Sasan, and Soheil Salehi. Automated hardware logic obfuscation framework using gpt. In *Dallas Circuits and Systems Conference (DCAS)*, 2024.

[210] Madhav Nair, Rajat Sadhukhan, Hammond Pearce, Debdeep Mukhopadhyay, and Ramesh Karri. Netlist whisperer: Ai and nlp fight circuit leakage! In *Workshop on Attacks and Solutions in Hardware Security (ASHES)*, 2023.

[211] Amisha Srivastava, Sanjay Das, Navnil Choudhury, Rafail Psiakis, Pedro Henrique Silva, Debjit Pal, and Kanad Basu. Scar: Power side-channel analysis at rtl level. *IEEE Transactions on Very Large Scale Integration Systems (TVLSI)*, 2024.

[212] Hammond Pearce, Benjamin Tan, Baleegh Ahmad, Ramesh Karri, and Brendan Dolan-Gavitt. Examining zero-shot vulnerability repair with large language models. In *2023 IEEE Symposium on Security and Privacy (SP)*, pages 2339–2356. IEEE, 2023.

[213] Georgios Kokolakis, Athanasios Moschos, and Angelos D Keromytis. Harnessing the power of general-purpose llms in hardware trojan design. In *International Conference on Applied Cryptography and Network Security (ACNS)*, 2024.

[214] Shams Tarek, Dipayan Saha, Sujan Kumar Saha, Mark Tehranipoor, and Farimah Farahmandi. Socurellm: An llm-driven approach for large-scale system-on-chip security verification and policy generation. *Cryptology ePrint Archive*, 2024.

[215] Yu-Zheng Lin, Muntasir Mamun, Muhtasim Alam Chowdhury, Shuyu Cai, Mingyu Zhu, Banafsheh Saber Latibari, Kevin Immanuel Gubbi, Najmeh Nazari Bavarsad, Arjun Caputo, Avesta Sasan, et al. Hw-v2w-map: Hardware vulnerability to weakness mapping framework for root cause analysis with gpt-assisted mitigation suggestion. *arXiv preprint arXiv:2312.13530*, 2023.

[216] Zhuolun He, Haoyuan Wu, Xinyun Zhang, Xufeng Yao, Su Zheng, Haisheng Zheng, and Bei Yu. Chateda: A large language model powered autonomous agent for eda. In *International Symposium on Machine Learning for CAD (MLCAD)*, 2023.

[217] Boyu Han, Xinyu Wang, Yifan Wang, Junyu Yan, and Yidong Tian. New interaction paradigm for complex eda software leveraging gpt. *arXiv preprint arXiv:2307.14740*, 2023.

[218] Shan Huang, Jinhao Li, Zhen Yu, Jiancai Ye, Jiaming Xu, Ningyi Xu, and Guohao Dai. Llsm: Llm-enhanced logic synthesis model with eda-guided cot prompting, hybrid embedding and aig-tailored acceleration. In *Asia and South Pacific Design Automation Conference (ASP-DAC)*, 2025.

[219] Manar Abdelatty, Jingxiao Ma, and Sherief Reda. Metrex: A benchmark for verilog code metric reasoning using llms. In *Asia and South Pacific Design Automation Conference (ASP-DAC)*, 2025.

[220] Chen-Chia Chang, Chia-Tung Ho, Yaguang Li, Yiran Chen, and Haoxing Ren. Drc-coder: Automated drc checker code generation using llm autonomous agent. *arXiv preprint arXiv:2412.05311*, 2024.

[221] Chenhui Deng, Yunsheng Bai, and Haoxing Ren. Chipalign: Instruction alignment in large language models for chip design via geodesic interpolation. *arXiv preprint arXiv:2412.19819*, 2024.

[222] Yuqi Jiang, Xudong Lu, Qian Jin, Qi Sun, Hanming Wu, and Cheng Zhuo. Fabgpt: An efficient large multimodal model for complex wafer defect knowledge queries. *arXiv preprint arXiv:2407.10810*, 2024.

[223] Guojin Chen, Haoyu Yang, Yu Bei, and Haoxing Ren. Intelligent opc engineer assistant for semiconductor manufacturing. In *AAAI Conference on Artificial Intelligence (AAAI)*, 2024.

[224] Chia-Tung Ho and Haoxing Ren. Large language model (llm) for standard cell layout design optimization. In *LLM Aided Design Workshop (LAD)*, 2024.

[225] Mengming Li, Wenji Fang, Qijun Zhang, and Zhiyao Xie. SpecLLM: Exploring generation and review of vlsi design specification with large language model. *arXiv preprint arXiv:2401.13266*, 2024.

[226] Zheyu Yan, Yifan Qin, Xiaobo Sharon Hu, and Yiyu Shi. On the viability of using LLMs for SW/HW co-design: An example in designing CiM DNN accelerators. *arXiv preprint arXiv:2306.06923*, 2023.

[227] Zhiding Liang, Jinglei Cheng, Rui Yang, Hang Ren, Zhixin Song, Di Wu, Xuehai Qian, Tongyang Li, and Yiyu Shi. Unleashing the potential of LLMs for quantum computing: A study in quantum architecture design. *arXiv preprint arXiv:2307.08191*, 2023.

[228] Yuan Pu, Zhuolun He, Tairu Qiu, Haoyuan Wu, and Bei Yu. Customized retrieval augmented generation and benchmarking for eda tool documentation qa. *arXiv preprint arXiv:2407.15353*, 2024.

[229] Chengjie Liu, Yijiang Liu, Yuan Du, and Li Du. Ladac: Large language model-driven auto-designer for analog circuits. *Authorea Preprints*, 2024.

[230] Yao Lai, Sungyoung Lee, Guojin Chen, Souradip Poddar, Mengkang Hu, David Z Pan, and Ping Luo. Analogcoder: Analog circuit design via training-free code generation. *arXiv preprint arXiv:2405.14918*, 2024.

[231] Yunwei Mao, You You, Xiaosi Tan, Yongming Huang, Xiaohu You, and Chuan Zhang. Flag: Formula-llm-based auto-generator for baseband hardware. In *International Symposium on Circuits and Systems (ISCAS)*, 2024.

[232] Yuxuan Yin, Yu Wang, Boxun Xu, and Peng Li. Ado-llm: Analog design bayesian optimization with in-context learning of large language models. *arXiv preprint arXiv:2406.18770*, 2024.

[233] Chen-Chia Chang, Yikang Shen, Shaoze Fan, Jing Li, Shun Zhang, Ningyuan Cao, Yiran Chen, and Xin Zhang. Lamagic: Language-model-based topology generation for analog integrated circuits. *arXiv preprint arXiv:2407.18269*, 2024.

[234] Zihao Chen, Jiangli Huang, Yiting Liu, Fan Yang, Li Shang, Dian Zhou, and Xuan Zeng. Artisan: Automated operational amplifier design via domain-specific large language model. In *Design Automation Conference (DAC)*, 2024.

[235] Dimple Vijay Kochar, Hanrui Wang, Anantha Chandrakasan, and Xin Zhang. Ledro: Llm-enhanced design space reduction and optimization for analog circuits. *arXiv preprint arXiv:2411.12930*, 2024.

[236] Haoyi Zhang, Shizhao Sun, Yibo Lin, Runsheng Wang, and Jiang Bian. Analogxpert: Automating analog topology synthesis by incorporating circuit design expertise into large language models. *arXiv preprint arXiv:2412.19824*, 2024.

[237] Jian Gao, Weidong Cao, Junyi Yang, and Xuan Zhang. Analoggenie: A generative engine for automatic discovery of analog circuit topologies. In *International Conference on Learning Representations (ICLR)*, 2025.

[238] Enrique Dehaerne, Bappaditya Dey, Sandip Halder, and Stefan De Gendt. A deep learning framework for verilog autocompletion towards design and verification automation. *arXiv preprint arXiv:2304.13840*, 2023.

[239] Zhuorui Zhao, Ruidi Qiu, Ing-Chao Lin, Grace Li Zhang, Bing Li, and Ulf Schlichtmann. Vrank: Enhancing verilog code generation from large language models via self-consistency. *arXiv preprint arXiv:2502.00028*, 2025.

[240] Laria Reynolds and Kyle McDonell. Prompt programming for large language models: Beyond the few-shot paradigm. In *Extended Abstracts of the Conference on Human Factors in Computing Systems (CHI EA)*, 2021.

[241] Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. React: Synergizing reasoning and acting in language models. In *International Conference on Learning Representations (ICLR)*, 2023.

[242] Daniel M Ziegler, Nisan Stiennon, Jeffrey Wu, Tom B Brown, Alec Radford, Dario Amodei, Paul Christiano, and Geoffrey Irving. Fine-tuning language models from human preferences. *arXiv preprint arXiv:1909.08593*, 2019.

[243] Shobha Vasudevan, David Sheridan, Sanjay Patel, David Tcheng, Bill Tuohy, and Daniel Johnson. Goldmine: Automatic assertion generation using data mining and static analysis. In *Design, Automation and Test in Europe Conference and Exhibition (DATE)*, 2010.

[244] Samuele Germiniani and Graziano Pravadelli. Harm: a hint-based assertion miner. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 2022.

[245] Zhuomin Chai, Yuxiang Zhao, Wei Liu, Yibo Lin, Runsheng Wang, and Ru Huang. Circuitnet: An open-source dataset for machine learning in vlsi cad applications with improved domain-specific evaluation metric and learning strategies. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 2023.

[246] Zeju Li, Changran Xu, Zhengyuan Shi, Zedong Peng, Yi Liu, Yunhao Zhou, Lingfeng Zhou, Chengyu Ma, Jianyuan Zhong, Xi Wang, et al. Deepcircuitx: A comprehensive repository-level dataset for rtl code understanding, generation, and ppa analysis. *arXiv preprint arXiv:2502.18297*, 2025.

[247] Shang Liu, Wenji Fang, Yao Lu, Qijun Zhang, and Zhiyao Xie. Towards big data in ai for eda research: Generation of new pseudo-circuits at rtl stage. In *Asia and South Pacific Design Automation Conference (ASP-DAC)*, 2025.

[248] Shang Liu, Jing Wang, Wenji Fang, and Zhiyao Xie. Syncircuit: Automated generation of new synthetic rtl circuits can enable big data in circuits. In *Design Automation Conference (DAC)*, 2025.

[249] Samuel Coward, Theo Drane, Emiliano Morini, and George A Constantinides. Combining power and arithmetic optimization via datapath rewriting. In *Symposium on Computer Arithmetic (ARITH)*, 2024.

[250] Chen Chen, Guangyu Hu, Dongsheng Zuo, Cunxi Yu, Yuzhe Ma, and Hongce Zhang. E-syn: E-graph rewriting with technology-aware cost functions for logic synthesis. In *Design Automation Conference (DAC)*, 2024.