Large Language Models (LLMs) for Verification, Testing, and Design

Chandan Kumar Jha[1], Muhammad Hassan[1,2], Khushboo Qayyum[2], Sallar Ahmadi-Pour[1],
Kangwei Xu[3], Ruidi Qiu[3], Jason Blocklove[4], Luca Collini[4], Andre Nakkab[4], Ulf Schlichtmann[3],
Grace Li Zhang[5], Ramesh Karri[4], Bing Li[6], Siddharth Garg[4], and Rolf Drechsler[1,2]
[1] Institute of Computer Science, University of Bremen, Germany
[2] Cyber-Physical Systems, DFKI GmbH, Germany
[3] Chair of Electronic Design Automation, Technical University of Munich, Germany
[4] Tandon School of Engineering, New York University, USA
[5] Hardware for Artificial Intelligence Group, Technical University of Darmstadt, Germany
[6] Research Group of Digital Integrated Systems, University of Siegen, Germany
{chajha, hassan, sallar, drechsler}@uni-bremen.de, khushboo.qayyum@dfki.de
{kangwei.xu, r.qiu, ulf.schlichtmann}@tum.de, grace.zhang@tu-darmstadt.de, bing.li@uni-siegen.de
{jblocklove,lc4976,ajn313,rkarri,sg175}@nyu.edu

***Abstract*—*Large Language Models* (LLMs) are being explored for their use in the domain of *Electronic Design Automation* (EDA). In this paper, we discuss state-of-the-art works showing the use of LLMs in verification, testing, and design generation. We provide a summary of the existing works and highlight the methods that have been used to enhance the quality of the output of the LLMs, like prompt engineering, *Retrieval Augmented Generation* (RAG), fine-tuning, multi-shot prompting, etc. We show that LLMs can aid in the domain of EDA, however, several challenges need to be addressed, such as data availability for fine-tuning the LLMs, integration with EDA tools, scalability, etc. This paper aims to highlight the use of LLMs in EDA, improve the output quality when using LLMs, and highlight the challenges and future directions that can be useful for further research.**

***Index Terms*—Large language models, formal verification, testing, Verilog, assertion based verification.**

I. Introduction

The capabilities of the *Large Language Models* (LLMs) are improving rapidly and they are being employed in a wide variety of applications [1], [2]. The ability of the LLMs to understand the context of data and generate human-like text is unparalleled [3]. LLMs achieve this capability by being trained on substantial amounts of data, which is both cost and resource-intensive [4], [5]. As a result, LLMs can produce a high-quality output for some of the tasks and can directly be used [6]. However, for complex domain-specific tasks, LLMs cannot be used directly as they are trained on massive datasets that are very generalized. Also, training the models from scratch for a particular application is not feasible as it would require a large time and cost, limiting the applicability of LLMs [7]. However, the key advantage these models have is that they are already pre-trained on vast amounts of data and only need to be fine-tuned on domain-specific datasets to be used readily [8], [9]. Another such method is *Retrieval Augmented Generation* (RAG), which can be used to improve the quality of the output generated by LLMs for domain-specific applications [10], [11]. RAG achieves this by providing more domain related context to the LLM. This allows the LLMs to be used in a wide variety of domains including *Electronic Design Automation* (EDA) [12]–[14].

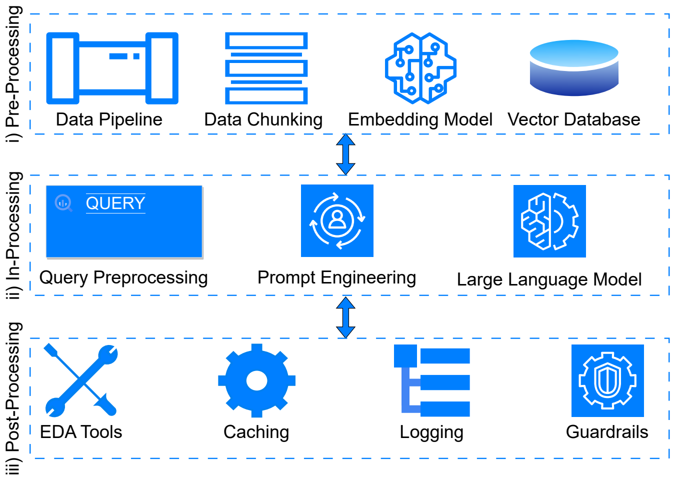

Fig. 1. Generalized LLM Application Architecture

In the domain of EDA, some areas in which the LLMs are employed are verification, testing, and design [15]–[17]. The use of LLMs in EDA ranges from directly using the web-based interface to building fully automated pipelines using frameworks like LangChain and Haystack [18], [19]. These frameworks ease the integration of LLMs into the development pipeline and can be used to either build proof-of-concept or entire applications. The approach to integrating LLMs to solve EDA problems in previous works varies widely given the range of problems that have been explored. However, there is a general approach to integrating the LLMs to help solve domain-specific problems.

A general application pipeline using LLMs is shown in Fig. 1. The LLM application architecture can be broken into three separate stages depending on when they appear in the pipeline. As mentioned earlier, while the LLMs are trained on a large set of data, for complex domain-specific tasks they may not give adequate quality results. While LLMs can be fine-tuned with domain-specific datasets which is also discussed later in the paper, this application architecture can be used without re-training to improve the output quality. The application architecture can be divided into three stages. i) Pre-processing, ii) In-processing, and iii) Post-processing.

i) Pre-processing: In the first stage the information related to the domain needs to be passed to the LLMs. Typically, the information is present in different formats like specification documents, pseudo codes, waveforms, libraries, etc. These must be converted into a format that can be understood by the LLM. Otherwise, asking LLM domain-specific questions can lead to hallucinations and incorrect responses from the LLMs [20], [21]. The conversion of entire documents cannot be done at once, as the LLMs have a limited context window, i.e., they only have context for up to a specified number of tokens. The data needs to be broken down into smaller segments of text, called data chunks. The data chunks are first processed by an embedding model, which converts them into vector representations (embeddings). These embeddings can then be used for similarity search or passed to a LLM for downstream tasks [22]. The vector embeddings are stored in the vector databases, which are used to provide more context related to domain-specific queries [23].

ii) In-Processing: In this stage, the LLMs are queried. The LLMs can be used without the pre-processing stage and will generate a response. However, to improve the quality of the response generated from the LLMs, the preprocessed vector database can now be used. The construction of the prompt heavily dictates the quality of the output generated from the LLM, hence, careful prompt construction is crucial [24]. The database in the pre-processing stage is queried to obtain relevant information. The query alongside the relevant information is crafted into a prompt, i.e., the input to the LLM. Various prompting techniques, such as chain-of-thought, multimodal prompts, etc., in addition to the results obtained from the EDA tools, can be further used to improve the quality of the output generated from the LLMs [25].

iii) Post-Processing: The output generated from the LLMs can be used directly or needs further processing depending upon the application. The output of the LLM can be fed to the EDA tools once or in an iterative manner, depending on the task. To improve the response time of the LLMs, caching can be used, i.e., the frequently accessed data is stored and accessed as needed [26]. The prompts and the responses can be effectively tracked using the logging tools like Promptlayer. Lastly, to tailor the quality of the output generated from the LLMs, guardrails can be used to limit the response of the LLM only to the provided context [27]. Guardrails can provide an additional layer of safety and consistency before the responses are generated and used.

These frameworks allow for the rapid development and adaptation of LLMs in EDA. In the following sections, we will discuss the use of LLMs for verification, testing, and design generation. While not all the stages of the LLM application architecture is used in one work, a selection of these are used and shows how LLMs can aid in the EDA domain. The paper is organized as follows. In Section II, we discuss the use of LLMs for verification. In Section III, we discuss the use of LLMs for testing. In Section IV, we discuss the use of LLMs for designing the hardware. Lastly, in Section V, we conclude the paper.

II. LLMs for Verification

Several works have explored machine learning techniques for verification [12], [28]. In this section, we aim to highlight the research problems in verification where LLMs in particular have shown potential. Even though the use of LLMs initially in the domain of EDA was limited to the generation of the hardware description code, it has significantly advanced thereafter in the area of verification.

In [29], is one of the first works in which the authors proposed using LLMs to aid in the verification process. The authors proposed a framework called *nl2sva*, that can generate *System Verilog Assertions* (SVAs) for a required circuit from a generic specification in natural language. The methodology is based on few-shot prompting and chain of thought examples were given to the LLM for improving the quality of the generated SVA. The human and the model checker are used to provide feedback to the LLM to aid in the debugging process. The authors state that given the limited context window of the LLMs, the approach cannot be extended for complex designs. Hence, the authors plan to look into methods that can extract necessary information from the circuit required for the generation of the SVAs.

In [30], the authors highlighted the difficulty of converting the specifications in natural language to assertions that can be used in the verification as it requires a lot of manual effort. To mitigate this, the authors showed that the process can be automated using the LLMs. However, rather than asking the LLMs to directly generate the specifications from the description files the authors proposed a three-step automated methodology, a) extraction of the information related to the generation of SVAs from the specification document, b) aligning the names in the natural language specification and the *Hardware Description Language* (HDL) implementation, and c) generation of the SVA. The authors were able to generate 89% of the SVA that were both structurally and functionally correct.

In [31], the authors propose a two-step methodology for generating SVAs for the Open Titan designs. In the first step, the design specifications were converted to a JSON file for being fed as a prompt to the LLM. The JSON file has a fixed formatting for the specification. In the second step, the SVAs generated from the LLM are checked for correctness using the VCS simulation tool. The log files of the simulation tool are used to correct the SVAs as needed iteratively. While less than 27% of the assertions required refinement, the authors comment that domain-specific LLM for *Assertion Based Verification* (ABV) would improve the quality of the generated assertions.

In [32], the authors reduced the number of iterations in the generation of the SVAs, by introducing a loop. In this loop, the Verilog design is fed alongside the specification to the LLM. This helps in the synchronization of the signal names in the design files and the SVAs. In addition to this manual prompting and the errors obtained from the simulator were used to refine the SVAs generated using the LLMs. The authors have suggested that *consistency* and *completeness* of the properties would be evaluated in the future [33].

In [34], the authors investigated the quality of the invariants generated using the LLMs. The invariants were generated by using the specification and the Verilog code using the LLMs. Mutations were generated in the design using the Yosys tool, and then the quality of the invariants was investigated. The authors also mentioned several additional insights that they observed during their experiments. LLMs work well a) with structured data, as also shown in [31], b) positive feedback and monetary incentives help generate better results from the LLMs, c) given the counterexample, the LLMs were unable to patch the design, d) the LLMs were unable to understand flattened netlists.

In [35], rather than using a general LLM, the authors propose the use of domain-specific models, i.e., a general model that is further trained on VLSI domain data to achieve better results. The authors use the syntax *pass@k* and the BLEU score as the metrics for comparison with existing LLMs. While their proposed approach was better than similar-sized models, the GPT-4 model outperformed every other LLM model in the SVA generation from the natural specification. The authors state that further improving the quality of results from the domain-adapted LLM requires training on formal verification-specific data.

In [36], the authors showed the methodology for patching bugs in the design using LLMs and RAG. The authors developed an iterative process for the detection and patching of bugs in the design. RAG was used to give more context to the LLM while prompting. The authors classified the five different types of bugs and introduced these into the Opentitan designs. LLM was able to patch the design for four types of these bugs, except for the case where the bug was related to an incorrect value of a constant.

In [37], the authors highlight the issue that the generated Verilog code using the LLMs has a lot of syntax errors and needs to be fixed. In addition to RAG, the authors use *React-based* prompting consisting of thought, action, and observation to improve the quality of correction of the Verilog code. It was shown that 98.5% syntax errors were corrected using their methodology. It was also shown that GPT-4 was able to achieve 98% syntax correction even with one-shot prompting. However, their approach can be used with smaller LLMs to improve the quality of the syntax correction.

In [38], the authors propose an iterative methodology for the correction of syntactical errors as well as functional errors. The authors use two LLM-based agents, one for debugging the *Register Transfer Level* (RTL) design and the other to score the quality of the RTL design in terms of completeness and overall quality. The authors improve the quality of the correction by fine-tuning the LLM and using prompting techniques like *self-planning* and *role prompting*.

In [39], the authors proposed using domain-specific LLM for the debugging of hardware. To train the LLM, the authors propose to use the design defects and their correction from the version control data of the open-source repositories to generate the training dataset. The authors then train a medium-sized LLM using this dataset for the detection and correction of bugs. The authors commented on the varying quality of results obtained using different fine-tuned models as a problem that needs to be addressed.

In [40], the authors proposed the first comprehensive benchmark suite in addition to an evaluation framework that tries to understand the capabilities of LLMs for the generation of SVA. The benchmark suite captures three different cases. First, *NL2SVA-Human*, checks the capability of the LLM to generate SVA, given a high-quality human-written testbench and high-level design specifications. Second, *NL2SVA-Machine*, focuses on whether the LLM can handle diverse specifications in natural language and generate the same SVA. Lastly, and the most challenging is the *Design2SVA* in which the capability of the LLM to generate the SVA only given the RTL design is checked. In [41], a comprehensive set of more than 1000 programs for *High Level Synthesis* (HLS) was selected. In each of the designs, up to 40 different bugs were injected. Hence, this dataset can be used to train the LLM for bug detection and correction.

In [42], the authors proposed the verification of RISC-V processors (traditional) and neuromorphic processors (domain specific) using LLMs. It was shown that LLMs have the potential for the verification of processors achieving up to 89% coverage. However, their methodology requires human intervention, making the scalability of their approach a challenge. The authors suggest that their methods can be made more scalable using automation, for example in the conversion of coverage results to prompts for the LLM.

In [43], the authors used the LLM to generate proofs that can be used for verification. The problem of proof generation was done in multiple steps. The code was initially split into smaller modules, then simple modules were identified, properties were generated for these simple modules, then the connections to these basic modules were identified, and the interconnection properties were generated. This approach of proof generation can be useful for designs that are implemented in a hierarchy, as the proof process can be iterated going up the hierarchy, eventually generating the proof for the entire design. In [44], the authors use the LLMs to generate helper properties which aid in the verification of the complex properties. The authors also used the counter-example generated during the induction-based verification step to generate assertions that aid during the k-induction-based proofs.

In all the above-mentioned works, the focus is on the correctness of the functionality of the designs. However, in recent years, the security of the designs has also become a serious concern. Some works are also exploring LLMs for the generation of assertions from the perspective of making the designs secure [45]–[47].

From the prior works, it is clear that the LLMs indeed have immense potential for use in hardware verification. Methods like training the LLMs, using RAG, and prompt engineering can be used to improve the quality of the results significantly. However, there are limited datasets that can be used for fine-tuning the LLMs for verification, and there is a need for the development of such datasets. Most of the data chunking strategies that are being used are the same as those used for the natural language. Different chunking strategies need to be explored to improve the quality of the LLMs when RAG is employed. Multimodal LLMs, i.e., rather than only using text as input, waveform files, state machine diagrams, etc., can further enhance the quality of the LLMs. In terms of the applications, the main focus of the use of LLMs has been on the correctness of the designs, other metrics like consistency and completeness need more exploration. Also, with the advancements of the LLMs, harder problems related to proof generation and deductive reasoning can be explored.

III. LLMs for Testing

Recently, the advent of LLMs demonstrates their superior capabilities and expertise in the field of EDA [49], [50], and their application in hardware testing is emerging as a particularly promising frontier. Although LLMs have proven effective in automated hardware testing, the complexity of testing tasks and the specialized nature of HDLs continue to pose significant challenges. In this section, a comprehensive review of recent advancements in LLM-aided hardware testing is provided to evaluate whether this integration is an actual breakthrough or an overestimated future.

A key step in these advancements is the use of interactive prompts and iterative improvements, which are essential for LLM-aided hardware testing. In [51], the authors use conversational prompts to produce testbenches that systematically verify the design functionality. By iteratively refining testbenches based on feedback from simulation tools, this method aims to achieve comprehensive functional coverage with minimal human intervention. Experimental results show that LLM-generated testbenches are effective in bug detection but still struggle to fully cover all aspects of the design and accurately reflect the intended behavior.

In [52], the authors introduce LLM4DV, a novel prompt template that interactively generates test stimuli by integrating four prompting techniques. Experimental results on three self-designed *Design Under Test* (DUT) modules show that LLM4DV outperforms conventional *Constrained Random Testing* (CRT) by effectively leveraging the LLM's inherent mathematical reasoning and extensive pre-trained knowledge.

Building on this foundation, the *LLM-Driven Test Generation* (LLM-TG) framework in [53] uses LLMs to interpret RTL behavior and construct effective prompts that produce high-quality test cases. Moreover, an open-source prompt library has been developed to standardize test generation for processor verification, thereby enhancing efficiency and reducing human intervention. Experimental results on a 12-stage, multi-issue, out-of-order RV64GC processor indicate that LLM-TG increases testing coverage effectively compared to the previous work [52].

In [54], the authors focus on enhancing the generation of testbenches, which are crucial for validating the functionality and achieving comprehensive coverage of RTL designs. By integrating iterative feedback from simulation tools into the testbench generation workflow, this framework accurately analyzes Verilog code to produce test stimuli that target previously uncovered branches and transitions in *Finite State Machine* (FSM). Experimental results demonstrate that the framework not only improves test coverage compared to random testing but also increases the precision in detecting functional discrepancies.

As conversational prompts and iterative improvements continue to evolve, testing methodologies are becoming increasingly essential. VerilogReader [55] leverages LLMs to reshape the *Coverage Driven Test Generation* (CDG) process. In this approach, the LLM acts as a 'Verilog Reader', accurately parsing the Verilog code to identify its underlying logic and then generating stimuli that activate previously untested code branches. Moreover, the coverage-explainer and design-under-test-explainer are also introduced to enrich prompts, thereby refining the generation of test stimuli. By comparing LLM4DV [52] with conventional CRT, the authors demonstrate that their framework significantly outperforms CRT within the LLM's comprehension range.

Fine-tuning large language models for hardware testing is also a promising and interesting direction. In [39], the authors pioneer this approach by adapting domain-specific models for automated bug detection and repair in hardware designs called LLM4SecHW. It generates a dataset of bugs from version histories, which is used to fine-tune medium-sized LLMs. For bug detection, the fine-tuned models analyze natural language prompts alongside HDL, identifying potential bugs by comparing the implementation to established design guidelines. In the subsequent bug repair, the system uses the same domain-specific knowledge to generate repair suggestions, refining HDL iteratively until all bugs are resolved. Experimental evaluations demonstrate that the framework can detect and repair bugs with high precision, significantly reducing the manual effort traditionally required in hardware testing.

Due to incomplete analysis that could lead to ambiguities in the LLM's understanding, step-wise reasoning becomes essential in hardware testing. In [56], the authors propose an LLM-aided approach to automating hardware testing by integrating hardware designs with progressive principles (What, Where, Why). This method enables the linear scaling of dataset volumes without relying on costly, expert-crafted prompts, thereby reducing manual labor significantly. By fine-tuning fifteen general-purpose LLMs on this enriched dataset, the

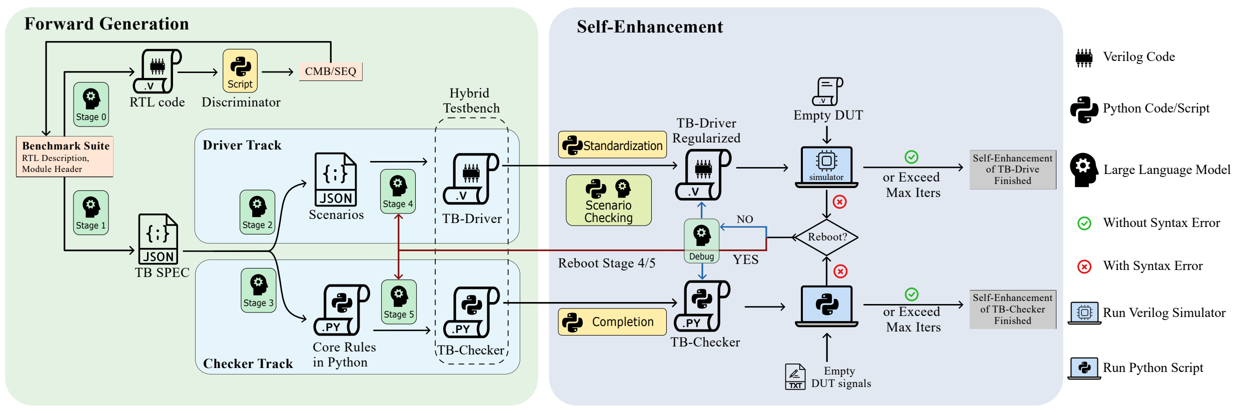

Fig. 2. AutoBench: Automatic testbench generation workflow for HDL design [48].

authors demonstrate a marked improvement in the models' capability to detect and repair bugs in hardware designs. The framework not only enhances reliability in test stimuli generation but also improves the overall accuracy of bug detection and repair.

The frontier of LLM-driven testing is also being further advanced in hardware-software interdisciplinary such as HLS. One notable advancement is the development of the Chrysalis dataset [57], which introduces various logical bugs into real-world HLS applications, creating a comprehensive benchmark that reflects bugs often overlooked by engineers. The dataset's construction involves a systematic process of HLS design collection, bug injection, and comprehensive validation, which assists in evaluating the efficiency of LLMs in bug detection, thereby simplifying HLS testing workflows. In addition, the paper discusses the quantization trade-off of LLMs, where reducing bit precision decreases memory usage and latency [58], but may lose accuracy.

In the following subsections, two case studies are presented to highlight the capabilities of LLMs in testbench generation.

A. LLM-Aided TestBench Generation for HDL Design

Simulation-based functional verification using a testbench is a crucial phase in the design of digital hardware. A testbench consists of two primary parts: the *driver* and the *checker*. The driver is responsible for generating test stimuli and directing the DUT to produce outputs. Subsequently, the checker captures these signals from the DUT and verifies the DUT's correctness. Traditional methods for testbench generation primarily automates the design of drivers with random test stimuli, while checkers remain manually designed due to their task-specific nature. Moreover, the use of randomly generated test stimuli can be inefficient during debugging since they lack additional context information.

Motivated by the significant performance of LLMs in circuit design, the first automatic and systematic testbench generation framework, AutoBench, was introduced in [48] to automates the design processes of both drivers and checkers, as illustrated in Fig. 2. The framework's only input is the RTL specification (SPEC) in natural language. Ultimately, it produces a hybrid testbench comprising both Python and Verilog codes. The workflow begins with identifying the circuit type from the given RTL SPEC, based on an imperfect RTL code generated by LLM using the RTL SPEC, as outlined in **Stage 0** of Fig. 2. Subsequently, in **Stage 1**, the RTL SPEC is translated into a TB SPEC by the LLM to support the subsequent stages.

1) Design of the driver: The driver code is responsible for generating test stimuli and directing the DUT. Here, the driver code comprises several test scenarios, each containing one or more test stimuli. In **Stage 2**, a list of test scenarios in natural language is produced by the LLM using the TB SPEC to facilitate comprehension in the subsequent stage. In **Stage 4**, the LLM is provided with both the SPEC and the scenario list to generate the complete testbench driver code in Verilog.

2) Design of the checker: The checker code in AutoBench is implemented in Python rather than Verilog due to Python's higher level of abstraction. Additionally, LLMs, leveraging a more extensive training dataset, exhibit superior performance in Python programming. The LLM generates the Python driver in two steps: it first creates the core functions in **Stage 3** and then produces the complete code in **Stage 5**.

3) Self-Enhancement: With the generated driver and checker, a self-enhancement process is undertaken to improve the testbench's correctness, as depicted in the right part of Fig. 2. Automatic code standardization and completion are performed according to predetermined formatting rules. For the checker, scenario checking is conducted to ensure that all scenarios in the list are present in the final driver code. Subsequently, both codes are executed to detect any syntax errors. The LLM attempts to resolve these errors using debugging information. If it fails after several attempts, the system reboots at the code generation process from **Stage 4** or **Stage 5**, depending on the code type.

The final revised and corrected codes will constitute the hybrid testbench produced by AutoBench. The implementation of AutoBench is accessible as open-source on https://github.com/AutoBench/AutoBench.

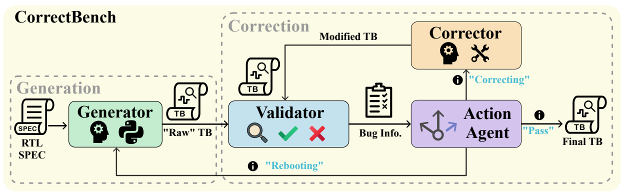

Fig. 3. CorrectBench: Automatic testbench generation workflow with self-correction [59].

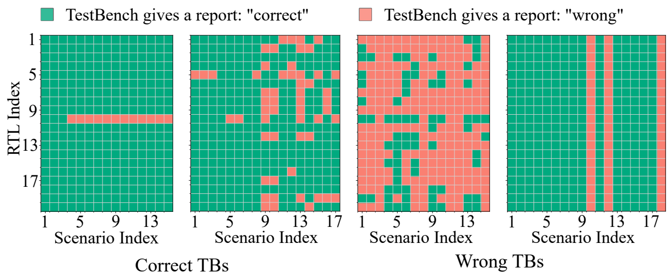

Fig. 4. Examples of RS Matrices.

B. *LLM-Aided TestBench Generation with Self-Correction*

Although achieving an average 57% improvement compared with directly generating testbench using LLMs, AutoBench still suffers from a low success rate. This limitation arises from the inherent uncertainty of LLMs, such as hallucination [60] and laziness [61]. Additionally, AutoBench employs only syntax self-checking, similar to RTLFixer [62], without implementing functional self-checking. The absence of a self-checking mechanism indeed limits the potential performance of the AutoBench framework.

To address the aforementioned issues, [59] proposed CorrectBench, the first framework for automatic testbench generation that incorporates functional self-validation and self-correction. The code implementation is also accessible as open-source on https://github.com/AutoBench/CorrectBench.

The framework of CorrectBench is drawn in Fig. 3. CorrectBench mainly focuses on the functional validation and correction of LLM-generated testbenches. Thus, the AutoBench [48], as mentioned in Section III-A and Fig. 2, is used as the testbench generator.

The generated testbench, called "raw" TB, is sent to the validator to do the functional validation (blue box in Fig. 3). The core part of CorrectBench is its validator, using an ensemble of imperfect RTLs to validate the correctness of generated testbenches from AutoBench. The testbench will generate a report for each RTL in each test scenario, which forms the RTL-Scenario matrix, as is depicted in Fig. 4. The more red blocks in one scenario, the more RTLs are marked as "wrong" by the testbench. Then, this test scenario is more likely to be wrong because the randomly distributed RTLs are likely to have errors in the same scenario.

After validation, a report with correct, wrong, and uncertain test scenario indexes (bug information), as well as the correctness of the testbench, is provided to the action agent (purple box). The action agent then decides one of the three actions as the next action: correcting such testbench with the corrector (orange box), rebooting the whole process, or ending it.

IV. LLMs for Design

LLMs often work best when working with natural language specifications, so the question quickly arose: "*Can LLMs reduce the burden on hardware designers by quickly mapping from a specification to a complete design?*" To this end, numerous works have explored the process of designing hardware using LLMs, either as assistants to human engineers or as fully autonomous systems.

A. *LLMs for Verilog Design*

The first paper to explore the use of LLMs to generate functional hardware descriptions was DAVE—a finetuned GPT-2 model trained on simple textbook-style Verilog problems [63]. The custom dataset used to train DAVE used a set of "Task/Result" problems, similar to those that would be taught as "novice" problems. The templates used were rather rigid, resulting in Verilog that was often correct, with a success rate of 94.8% across its validation tests, but lacked creativity or complexity. DAVE was not able to generate multiple styles of response to a task and did not create additional signals that were not provided in the request. This work was also limited by the LLMs of the time, with GPT-2 being significantly underpowered by current state-of-the-art standards.

In [64], the authors propose VeriGen, which is built upon the work from DAVE, by expanding the training dataset significantly as well as leveraging a more complex LLM. VeriGen consisted of five different fine-tuned LLMs, with the CodeGen family of models being the largest and most capable. These models were trained using a corpus of both open-source Verilog designs queried from GitHub and Verilog textbooks. The finetuned VeriGen models were shown to be similarly capable to the recently released at the time ChatGPT-3.5 and -4 models for small and medium problems, though GPT-4 was more capable with increasing complexity.

In [65], the authors propose VerilogEval, which is another finetuned LLM for generating Verilog, that followed after VeriGen and tuned the same CodeGen models. VerilogEval leveraged supervised fine-tuning on both Verilog extracted from GitHub and a set of synthetic data created by analyzing modules and using GPT-3.5 to generate functional descriptions of the modules. Similarly, RTLCoder [66] is another more modern family of finetuned LLMs aimed at generating Verilog, this time finetuning over Mistral and DeepSeek models. Both VerilogEval and RTLCoder are fully open-sourced models that claim to give higher rates of success than the major pre-trained models at their time of publishing, as well as VeriGen.

Finally, in [67], the authors propose CodeV that provides a series of new open-source instruction-tuned models for generating Verilog. These models differ from the previous offerings by tuning the base LLMs using summarized descriptions of the modules as well as the Verilog code itself. Thus far, CodeV claims to have the highest level of success on the

VerilogEval series of benchmarks, discussed in greater detail in Section IV-D

B. General Knowledge LLMs for Verilog

In [68], the authors propose Chip-Chat an experienced hardware engineer leveraged ChatGPT-4, at the time the most modern commercially available LLM, to assist in the specification, design, implementation, and testing of a novel 8-bit accumulator-based microprocessor architecture, which was ultimately taped out using the Skywater 130nm process. ChatGPT was prompted only using natural language and was responsible for all of the HDL used in the design of this architecture, as well as a Python assembler compatible with its custom *Instruction Set Architecture* (ISA). The conversations responsible for creating this design were unstructured, but different conversations were leveraged for different aspects of the design, such as "Register specification," "Control signal planning," and "ALU optimization." Chip-Chat demonstrated that, while not inherently capable of formulating a complete design of this nature on its own, an LLM could act as a significant force-multiplier in the hands of an experienced engineer, enabling them to use their time more efficiently for complex tasks and allowing the LLM to handle much of the actual HDL development.

Another work sent for tapeout was [52], in which ChatGPT-3.5 and -4 were prompted using a strict conversational script, aimed at developing both a simple design and testbench with minimal human intervention. Icarus Verilog was used to compile and simulate the designs and their testbenches, and the amount of user explanation needed to achieve functional designs was tracked. Eight different designs were prompted for, each three times, ranging from a simple shift register to state machines and a simple simulated dice roller—designs chosen to mimic designs that would be expected of undergraduate students taking a digital hardware design class. The initial set of designs generated by ChatGPT-4 were sent for tapeout on the same TinyTapeout shuttle as the processor in Chip-Chat, and came back functioning as expected, once again with no human-written HDL in the design. Prompting for both design and testbench did show some issues in LLMs' abilities to understand the specific functionality of the design being generated, however, as some testbenches could be made for which the design would pass all test cases, but still have incorrect functionality.

Several commercial hardware design-focused LLMs have been released as well, each claiming to have different specialties and capabilities. These include Nvidia's ChipNemo [69], Cadence's JedAI [70], Synopsys' Copilot [71], PrimisAI's RapidGPT [72].

C. Hardware Design with HLS

Verilog is not the only language able to be used to design hardware, and so is not the only language LLMs are being used for. In [73], the authors propose C2HLSC, which leverages LLMs to convert non-synthesizable C designs into HLS-compatible code. The framework for this conversion is shown in Fig. 5.

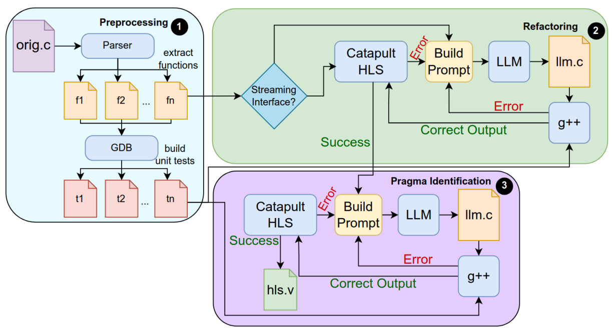

Fig. 5. C2HLSC framework [73]

The flow is divided into three steps. The first step does not use LLMs and its purpose is to set up the testing environment so that the LLM's outputs can be tested for functional correctness. Steps 2 and 3 work on a single function at a time, in a bottom up approach. This allows the flow to work with hierarchical designs, focusing on one function at a time. To avoid the users the burden of providing unit tests for each function, step one builds unit tests for each function starting from a top level test. Steps 2 and 3 work in a similar fashion: a feedback loop is used to prompt the LLM iteratively, providing feedback on its output. Step 2 has the goal of fixing synthesis errors, while Step 3 has the goal of inserting pragmas for optimization. The output of the LLM is first compiled and the obtained binary is executed to test for correct functionality using the test provided by step 1. Then the code is synthesized using the HLS tool. If an error arises from any of these substeps, a prompt asking to fix the error is generated and sent to the LLM.

D. Evaluating/Benchmarking LLMs

Quantifying the design abilities of an LLM in a standard and complete manner has remained a challenge since the first works on the subject began. VeriGen [74] proposed a set of 17 problems ranked as basic, intermediate, or advanced. These ranged from generating a simple wire or gate up to state machines and shift registers. A very similar subset of these problems was used in [52], but due to the I/O and clock speed restrictions of taping out with TinyTapeout3 only eight problems were given and were restricted in their complexity. These were, however, not intended to be used as global metrics for success.

In [75], the authors proposed RTLLM, where the first notable effort in creating an LLM-specific set of benchmarks that could be used to evaluate the design capabilities of LLMs. This included a total of 30 open-source benchmark designs, focusing on both arithmetic operations and logical operations, and varying significantly in the size and complexity of the designs. Simpler designs, such as a 0 to 12 counter, would require few lines of code, while more complex designs, such as an asynchronous FIFO, could require well over 100 lines

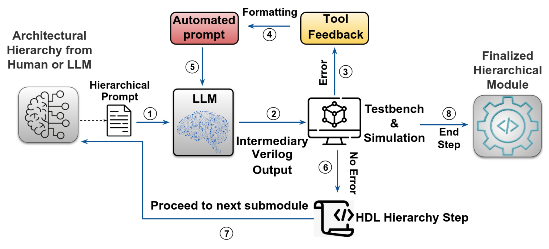
Architectural Hierarchy from Human or LLM
Hierarchical Prompt
Automated prompt
Formatting
Tool Feedback
LLM
Intermediary Verilog Output
Error
No Error
Testbench & Simulation
End Step
Finalized Hierarchical Module
Proceed to next submodule
HDL Hierarchy Step

Fig. 6. Automatic hierarchical framework [77]

of HDL. The most complex benchmark was a simplified RISC CPU, which required over 500 lines of code, which could not able to be done by any of the available LLMs at the time of its release.

In [76], the authors proposed VerilogEval, which sought to expand the size of the benchmark set significantly by using the problems from the Verilog teaching website HDLBits. While these benchmarks were originally used with VeriGen, they were further formalized in VerilogEval and became available with complete testbenches for evaluation. 156 different problems were made available with this benchmark set, allowing for a significant level of additional evaluation of generated designs. However, the HDLBits problems were simple, and all were single-module designs. A second version of the VerilogEval benchmark set was released, which simplified the methods of using the benchmarks and added more complex benchmarks that tested for more aspects of design.

E. Automated Design with LLMs

Most of the previously mentioned LLMs and frameworks are designed to be used by a human engineer to aid in the design process, however, several works have sought to remove the human from the loop and enable LLMs given only an initial setup to generate complete functional designs.

In [77], the authors propose CL-Verilog, which is the most recent Verilog-specific LLM, fine-tuned over Code Llama. Along with the CL-Verilog model, a framework for automatically generating large hierarchical designs based on smaller designs was proposed, which enables the generation of designs more realistic than those that would be made in practical applications, such as small cryptographic accelerators or systolic arrays. This framework is shown in Fig. 6.

This automatic Verilog generation framework aims specifically at creating larger hierarchical designs by first extracting a list of necessary submodules from the design description, then building each submodule, and finally integrating them into the top-level of the design. Using this structure, along with a set of specifically hierarchical benchmarks, CL-Verilog consistently had the highest rate of success, completing most problems successfully within 5 attempts.

In [78], the authors propose AutoChip, which seeks to use feedback from EDA tools to automate the process of generating, evaluating, and further iterating upon a design given just a natural language specification and a testbench. This is accomplished using a tree-search methodology, by which

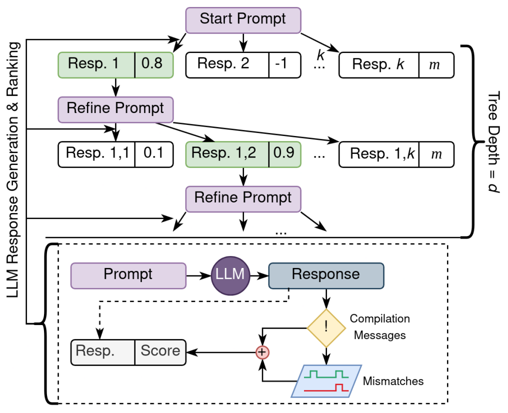

Fig. 7. AutoChip tree search framework [78]

multiple candidate responses are generated and evaluated using Icarus Verilog and an included testbench, the best of these responses is then iterated upon until either a correct module is generated or a stopping criteria is met—tree depth, total token cost, total number of queries, etc. This framework is shown in Fig. 7.

AutoChip was evaluated on four commercially available chat-based LLMs using the set of benchmarks from VerilogEval v1. It was found that the selection of the model had a significant impact on the usefulness of tool feedback, with it being more successful and computationally efficient to request more candidate responses in a zero-shot manner than to rely on the tree search. The tree search methodology did help with GPT-4o; however, as GPT-4o seemed better able to interpret the errors being reported from EDA tools and failed tests.

V. Conclusion and Future Works

In this paper, we discussed the use of LLMs in verification, testing, and generation. We further discussed a generalized architecture used for integrating the LLMs in the domain of EDA. We showed how general LLM can be tailored for use in the domain of EDA. While fine-tuning is one of the strategies, methods like RAG can be used to provide the LLM with domain-specific context. To improve the adaptability of LLMs, domain-specific datasets are required to perform fine-tuning. For verification problems, consistency and coverage goals still need significant improvement. For testing, the integration of coverage-oriented self-correction during testbench generation can be further explored. Real-time test strategies can be explored for *High-level Synthesis* (HLS) testing. For design, further exploration needs to be done in using feedback to improve responses and further leveraging LLMs for alternate hardware design methods like HLS. In general, by developing adaptive prompt strategies and domain-specific fine-tuning protocols, models can be better tailored to the nuances of hardware design.

Acknowledgements

This work was supported in part by the German Federal Ministry of Education and Research (BMBF) within the project ECXL under contract no. 01IW22002, the project ECXLplus under contract no. 01IW24001, the project Scale4Edge under contract no. 16ME0127, and the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) – Project-ID 504518248. Portions of this work were also supported by grants from Synopsys, DTCC, and the NSF.

REFERENCES

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

[2] META, "Llama models." [Online]. Available: https://github.com/meta-llama/llama

[3] J. Wei, Y. Tay, R. Bommasani, C. Raffel, B. Zoph, S. Borgeaud, D. Yogatama, M. Bosma, D. Zhou, D. Metzler *et al.*, "Emergent abilities of large language models," *Transactions on Machine Learning Research*, 2022.

[4] T. Wu, S. He, J. Liu, S. Sun, K. Liu, Q.-L. Han, and Y. Tang, "A brief overview of chatgpt: The history, status quo and potential future development," *IEEE/CAA Journal of Automatica Sinica*, vol. 10, no. 5, pp. 1122–1136, 2023.

[5] A. Liu, B. Feng, B. Xue, B. Wang, B. Wu, C. Lu, C. Zhao, C. Deng, C. Zhang, C. Ruan *et al.*, "Deepseek-v3 technical report," *arXiv preprint arXiv:2412.19437*, 2024.

[6] X. Chen, C. Gao, C. Chen, G. Zhang, and Y. Liu, "An empirical study on challenges for llm application developers," *ACM Transactions on Software Engineering and Methodology*, 2025.

[7] L. Zhang, X. Liu, Z. Li, X. Pan, P. Dong, R. Fan, R. Guo, X. Wang, Q. Luo, S. Shi *et al.*, "Dissecting the runtime performance of the training, fine-tuning, and inference of large language models," *arXiv preprint arXiv:2311.03687*, 2023.

[8] Y. Xia, J. Kim, Y. Chen, H. Ye, S. Kundu, C. C. Hao, and N. Talati, "Understanding the performance and estimating the cost of llm fine-tuning," in *2024 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, 2024, pp. 210–223.

[9] X. Lin, W. Wang, Y. Li, S. Yang, F. Feng, Y. Wei, and T.-S. Chua, "Data-efficient fine-tuning for llm-based recommendation," in *Proceedings of the 47th international ACM SIGIR conference on research and development in information retrieval*, 2024, pp. 365–374.

[10] Y. Gao, Y. Xiong, X. Gao, K. Jia, J. Pan, Y. Bi, Y. Dai, J. Sun, H. Wang, and H. Wang, "Retrieval-augmented generation for large language models: A survey," *arXiv preprint arXiv:2312.10997*, vol. 2, 2023.

[11] J. J. Pan, J. Wang, and G. Li, "Vector database management techniques and systems," in *Companion of the 2024 International Conference on Management of Data*, 2024, pp. 597–604.

[12] R. Zhong, X. Du, S. Kai, Z. Tang, S. Xu, H.-L. Zhen, J. Hao, Q. Xu, M. Yuan, and J. Yan, "Llm4eda: Emerging progress in large language models for electronic design automation," *arXiv preprint arXiv:2401.12224*, 2023.

[13] K. Qayyum, S. Ahmadi-Pour, C. K. Jha, M. Hassan, and R. Drechsler, "Llms for hardware verification: Frameworks, techniques, and future directions," in *2024 IEEE 33rd Asian Test Symposium (ATS)*, 2024, pp. 1–6.

[14] Z. He and B. Yu, "Large language models for eda: Future or mirage?" in *Proceedings of the 2024 International Symposium on Physical Design*, 2024, pp. 65–66.

[15] R. Drechsler, *Advanced formal verification*. Springer, 2004.

[16] M. Bushnell and V. Agrawal, *Essentials of electronic testing for digital, memory and mixed-signal VLSI circuits*. Springer Science & Business Media, 2004, vol. 17.

[17] S. Palnitkar, *Verilog HDL: a guide to digital design and synthesis*. Prentice Hall Professional, 2003, vol. 1.

[18] LangChain, "Langchain." [Online]. Available: https:https://www.langchain.com/

[19] Haystack, "Haystack." [Online]. Available: https://haystack.deepset.ai/

[20] F. Liu, Y. Liu, L. Shi, H. Huang, R. Wang, Z. Yang, L. Zhang, Z. Li, and Y. Ma, "Exploring and evaluating hallucinations in llm-powered code generation," *arXiv preprint arXiv:2404.00971*, 2024.

[21] L. Huang, W. Yu, W. Ma, W. Zhong, Z. Feng, H. Wang, Q. Chen, W. Peng, X. Feng, B. Qin *et al.*, "A survey on hallucination in large language models: Principles, taxonomy, challenges, and open questions," *ACM Transactions on Information Systems*, vol. 43, no. 2, pp. 1–55, 2025.

[22] J. Xian, T. Teofili, R. Pradeep, and J. Lin, "Vector search with openai embeddings: Lucene is all you need," in *Proceedings of the 17th ACM International Conference on Web Search and Data Mining*, 2024, pp. 1090–1093.

[23] J. J. Pan, J. Wang, and G. Li, "Survey of vector database management systems," *The VLDB Journal*, vol. 33, no. 5, pp. 1591–1615, 2024.

[24] J. D. Zamfirescu-Pereira, R. Y. Wong, B. Hartmann, and Q. Yang, "Why johnny can't prompt: how non-ai experts try (and fail) to design llm prompts," in *Proceedings of the 2023 CHI conference on human factors in computing systems*, 2023, pp. 1–21.

[25] S. Schulhoff, M. Ilie, N. Balepur, K. Kahadze, A. Liu, C. Si, Y. Li, A. Gupta, H. Han, S. Schulhoff *et al.*, "The prompt report: A systematic survey of prompting techniques," *arXiv preprint arXiv:2406.06608*, 2024.

[26] F. Bang, "Gptcache: An open-source semantic cache for llm applications enabling faster answers and cost savings," in *Proceedings of the 3rd Workshop for Natural Language Processing Open Source Software (NLP-OSS 2023)*, 2023, pp. 212–218.

[27] T. Rebedea, R. Dinu, M. Sreedhar, C. Parisien, and J. Cohen, "Nemo guardrails: A toolkit for controllable and safe llm applications with programmable rails," *arXiv preprint arXiv:2310.10501*, 2023.

[28] N. Wu, Y. Li, H. Yang, H. Chen, S. Dai, C. Hao, C. Yu, and Y. Xie, "Survey of machine learning for software-assisted hardware design verification: Past, present, and prospect," *ACM Transactions on Design Automation of Electronic Systems*, vol. 29, no. 4, pp. 1–42, 2024.

[29] C. Sun, C. Hahn, and C. Trippel, "Towards improving verification productivity with circuit-aware translation of natural language to systemverilog assertions," in *First International Workshop on Deep Learning-aided Verification*, 2023.

[30] W. Fang, M. Li, M. Li, Z. Yan, S. Liu, H. Zhang, and Z. Xie, "Assertllm: Generating hardware verification assertions from design specifications via multi-llms," in *2024 IEEE LLM Aided Design Workshop (LAD)*. IEEE, 2024, pp. 1–1.

[31] B. Mali, K. Maddala, V. Gupta, S. Reddy, C. Karfa, and R. Karri, "Chiraag: Chatgpt informed rapid and automated assertion generation," in *2024 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*. IEEE, 2024, pp. 680–683.

[32] K. Maddala, B. Mali, and C. Karfa, "Laag-rv: Llm assisted assertion generation for rtl design verification," in *2024 IEEE 8th International Test Conference India (ITC India)*. IEEE, 2024, pp. 1–6.

[33] R. Drechsler, M. Diepenbeck, D. Große, U. Kühne, H. M. Le, J. Seiter, M. Soeken, and R. Wille, "Completeness-driven development," in *International Conference on Graph Transformation*. Springer, 2012, pp. 38–50.

[34] M. Hassan, S. Ahmadi-Pour, K. Qayyum, C. K. Jha, and R. Drechsler, "Llm-guided formal verification coupled with mutation testing," in *2024 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2024, pp. 1–2.

[35] M. Liu, M. Kang, G. B. Hamad, S. Suhaib, and H. Ren, "Domain-adapted llms for vlsi design and verification: A case study on formal verification," in *2024 IEEE 42nd VLSI Test Symposium (VTS)*. IEEE, 2024, pp. 1–4.

[36] K. Qayyum, M. Hassan, S. Ahmadi-Pour, C. K. Jha, and R. Drechsler, "From bugs to fixes: Hdl bug identification and patching using llms and rag," in *2024 IEEE LLM Aided Design Workshop (LAD)*. IEEE, 2024, pp. 1–5.

[37] Y. Tsai, M. Liu, and H. Ren, "Rtlfixer: Automatically fixing rtl syntax errors with large language model," in *Proceedings of the 61st ACM/IEEE Design Automation Conference*, 2024, pp. 1–6.

[38] K. Xu, J. Sun, Y. Hu, X. Fang, W. Shan, X. Wang, and Z. Jiang, "Meic: Re-thinking rtl debug automation using llms," *arXiv preprint arXiv:2405.06840*, 2024.

[39] W. Fu, K. Yang, R. G. Dutta, X. Guo, and G. Qu, "Llm4sechw: Leveraging domain-specific large language model for hardware debugging," in *2023 Asian Hardware Oriented Security and Trust Symposium (AsianHOST)*. IEEE, 2023, pp. 1–6.

[40] M. Kang, M. Liu, G. B. Hamad, S. Suhaib, and H. Ren, "Fveval: Understanding language model capabilities in formal verification of digital hardware," *arXiv preprint arXiv:2410.23299*, 2024.

[41] L. J. Wan, Y. Huang, Y. Li, H. Ye, J. Wang, X. Zhang, and D. Chen, "Software/hardware co-design for llm and its application for design verification," in *2024 29th Asia and South Pacific Design Automation Conference (ASP-DAC)*. IEEE, 2024, pp. 435–441.

[42] C. Xiao, Y. Deng, Z. Yang, R. Chen, H. Wang, J. Zhao, H. Dai, L. Wang, Y. Tang, and W. Xu, "Llm-based processor verification: A case study for neuromorphic processor," in *2024 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2024, pp. 1–6.

[43] K. Qayyum, M. Hassan, S. Ahmadi-Pour, C. K. Jha, and R. Drechsler, "Late breaking results: Llm-assisted automated incremental proof generation for hardware verification," in *Proceedings of the 61st ACM/IEEE Design Automation Conference*, 2024, pp. 1–2.

[44] A. Kumar and D. N. Gadde, "Generative ai augmented induction-based formal verification," in *2024 IEEE 37th International System-on-Chip Conference (SOCC)*. IEEE, 2024, pp. 1–2.

[45] S. S. Miftah, A. Srivastava, H. Kim, and K. Basu, "Assert-o: Context-based assertion optimization using llms," in *Proceedings of the Great Lakes Symposium on VLSI 2024*, 2024, pp. 233–239.

[46] R. Kande, H. Pearce, B. Tan, B. Dolan-Gavitt, S. Thakur, R. Karri, and J. Rajendran, "(security) assertions by large language models," *IEEE Transactions on Information Forensics and Security*, 2024.

[47] A. Ayalasomayajula, R. Guo, J. Zhou, S. K. Saha, and F. Farahmandi, "Lasp: Llm assisted security property generation for soc verification," in *Proceedings of the 2024 ACM/IEEE International Symposium on Machine Learning for CAD*, 2024, pp. 1–7.

[48] R. Qiu, G. L. Zhang, R. Drechsler, U. Schlichtmann, and B. Li, "AutoBench: Automatic Testbench Generation and Evaluation Using LLMs for HDL Design," in *ACM/IEEE International Symposium on Machine Learning for CAD (MLCAD)*, 2024.

[49] K. Xu, R. Qiu, Z. Zhao, G. L. Zhang, U. Schlichtmann, and B. Li, "Llm-aided efficient hardware design automation," 2024. [Online]. Available: https://arxiv.org/abs/2410.18582

[50] K. Xu, G. L. Zhang, X. Yin, C. Zhuo, U. Schlichtmann, and B. Li, "Automated c/c++ program repair for high-level synthesis via large language models," in *ACM/IEEE International Symposium on Machine Learning for CAD (MLCAD)*, 2024.

[51] Z. Zhang, G. Chadwick, H. McNally, Y. Zhao, and R. Mullins, "Llm4dv: Using large language models for hardware test stimuli generation," 2023. [Online]. Available: https://arxiv.org/abs/2310.04535

[52] J. Blocklove, S. Garg, R. Karri, and H. Pearce, "Evaluating llms for hardware design and test," in *2024 IEEE LLM Aided Design Workshop (LAD)*. IEEE, Jun. 2024, p. 1–6. [Online]. Available: http://dx.doi.org/10.1109/LAD62341.2024.10691811

[53] Y. Deng, R. Chen, C. Xiao, Z. Yang, Y. Luo, J. Zhao, N. Li, Z. Wan, Y. Ai, H. Dai, and L. Wang, "Llm - tg: Towards automated test case generation for processors using large language models," in *2024 IEEE 42nd International Conference on Computer Design (ICCD)*, 2024, pp. 389–396.

[54] J. Bhandari, J. Knechtel, R. Narayanaswamy, S. Garg, and R. Karri, "Llm-aided testbench generation and bug detection for finite-state machines," 2024. [Online]. Available: https://arxiv.org/abs/2406.17132

[55] R. Ma, Y. Yang, Z. Liu, J. Zhang, M. Li, J. Huang, and G. Luo, "Verilogreader: Llm-aided hardware test generation," in *2024 IEEE LLM Aided Design Workshop (LAD)*. IEEE, Jun. 2024, p. 1–5. [Online]. Available: http://dx.doi.org/10.1109/LAD62341.2024.10691801

[56] W. Fu, S. Li, Y. Zhao, K. Yang, X. Zhang, Y. Jin, and X. Guo, "A generalize hardware debugging approach for large language models semi-synthetic, datasets," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 72, pp. 623–636, 2025. [Online]. Available: https://api.semanticscholar.org/CorpusID:269742384

[57] L. J. Wan, Y. Huang, Y. Li, H. Ye, J. Wang, X. Zhang, and D. Chen, "Invited paper: Software/hardware co-design for llm and its application for design verification," in *2024 29th Asia and South Pacific Design Automation Conference (ASP-DAC)*, 2024, pp. 435–441.

[58] K. Xu, G. L. Zhang, U. Schlichtmann, and B. Li, "Logic design of neural networks for high-throughput and low-power applications," in *ACM/IEEE Proceedings of the 29th Asia and South Pacific Design Automation Conference (ASP-DAC)*, 2024.

[59] R. Qiu, G. L. Zhang, R. Drechsler, U. Schlichtmann, and B. Li, "CorrectBench: Automatic Testbench Generation with Functional Self-Correction using LLMs for HDL Design," *arXiv preprint: 2411.08510*, 2024.

[60] L. Huang, W. Yu, W. Ma, W. Zhong, Z. Feng, H. Wang, Q. Chen, W. Peng, X. Feng, B. Qin, and T. Liu, "A Survey on Hallucination in Large Language Models: Principles, Taxonomy, Challenges, and Open Questions," *arXiv preprint: 2311.05232*, 2023.

[61] R. Tang, D. Kong, L. Huang, and H. Xue, "Large Language Models Can be Lazy Learners: Analyze Shortcuts in In-Context Learning," in *Findings of the Association for Computational Linguistics: ACL 2023*, 2023.

[62] Y. Tsai, M. Liu, and H. Ren, "RTLFixer: Automatically Fixing RTL Syntax Errors with Large Language Models," *arXiv preprint: 2311.16543*, 2023.

[63] H. Pearce, B. Tan, and R. Karri, "DAVE: Deriving Automatically Verilog from English," in *Proceedings of the 2020 ACM/IEEE Workshop on Machine Learning for CAD*. Virtual Event Iceland: ACM, Nov. 2020, pp. 27–32. [Online]. Available: https://dl.acm.org/doi/10.1145/3380446.3430634

[64] S. Thakur, B. Ahmad, H. Pearce, B. Tan, B. Dolan-Gavitt, R. Karri, and S. Garg, "VeriGen: A Large Language Model for Verilog Code Generation," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 29, no. 3, pp. 46:1–46:31, Apr. 2024. [Online]. Available: https://doi.org/10.1145/3643681

[65] M. Liu, N. Pinckney, B. Khailany, and H. Ren, "VerilogEval: Evaluating Large Language Models for Verilog Code Generation," Dec. 2023, arXiv:2309.07544 [cs]. [Online]. Available: http://arxiv.org/abs/2309.07544

[66] S. Liu, W. Fang, Y. Lu, Q. Zhang, H. Zhang, and Z. Xie, "RTLCoder: Outperforming GPT-3.5 in Design RTL Generation with Our Open-Source Dataset and Lightweight Solution," Feb. 2024, arXiv:2312.08617 [cs]. [Online]. Available: http://arxiv.org/abs/2312.08617

[67] Y. Zhao, D. Huang, C. Li, P. Jin, Z. Nan, T. Ma, L. Qi, Y. Pan, Z. Zhang, R. Zhang, X. Zhang, Z. Du, Q. Guo, X. Hu, and Y. Chen, "Codev: Empowering llms for verilog generation through multi-level summarization," 2024. [Online]. Available: https://arxiv.org/abs/2407.10424

[68] J. Blocklove, S. Garg, R. Karri, and H. Pearce, "Chip-Chat: Challenges and Opportunities in Conversational Hardware Design," in *2023 ACM/IEEE 5th Workshop on Machine Learning for CAD (MLCAD)*, Sep. 2023, pp. 1–6. [Online]. Available: https://ieeexplore.ieee.org/document/10299874

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

[70] Cadence, "Cadence JedAI Generative AI Solution for Chip, System, and Product Design," Sep. 2023. [Online]. Available: https://www.cadence.com/en_US/home/solutions/joint-enterprise-data-ai-platform.html

[71] "Introducing Generative AI for Chip Design | Synopsys Blog." [Online]. Available: https://www.synopsys.com/blogs/chip-design/copilot-generative-ai-chip-design.html

[72] RapidSilicon, "RapidGPT," 2023. [Online]. Available: https://rapidsilicon.com/rapidgpt/

[73] L. Collini, S. Garg, and R. Karri, "C2hlsc: Can llms bridge the software-to-hardware design gap?" in *2024 IEEE LLM Aided Design Workshop (LAD)*, 2024, pp. 1–12.

[74] S. Thakur, B. Ahmad, Z. Fan, H. Pearce, B. Tan, R. Karri, B. Dolan-Gavitt, and S. Garg, "Benchmarking large language models for automated verilog rtl code generation," in *2023 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2023, pp. 1–6.

[75] Y. Lu, S. Liu, Q. Zhang, and Z. Xie, "RTLLM: An Open-Source Benchmark for Design RTL Generation with Large Language Model," Nov. 2023, arXiv:2308.05345 [cs]. [Online]. Available: http://arxiv.org/abs/2308.05345

[76] N. Pinckney, C. Batten, M. Liu, H. Ren, and B. Khailany, "Revisiting verilogeval: A year of improvements in large-language models for hardware code generation," 2025. [Online]. Available: https://arxiv.org/abs/2408.11053

[77] A. Nakkab, S. Q. Zhang, R. Karri, and S. Garg, "Rome was not built in a single step: Hierarchical prompting for llm-based chip design," in *2024 ACM/IEEE 6th Symposium on Machine Learning for CAD (MLCAD)*, 2024, pp. 1–11.

[78] J. Blocklove, S. Thakur, B. Tan, H. Pearce, S. Garg, and R. Karri, "Automatically improving llm-based verilog generation using eda tool feedback," 2025. [Online]. Available: https://arxiv.org/abs/2411.11856