Data is all you need: Finetuning LLMs for chip design via an automated design-data augmentation framework

Kaiyan Chang[1,3], Kun Wang[9,2], Nan Yang[2,3], Ying Wang✉[2], Dantong Jin[12], Wenlong Zhu[2,3], Zhirong Chen[6,10], Cangyuan Li[2,3], Hao Yan[7,5], Yunhao Zhou[7,11], Zhuoliang Zhao[7,8], Yuan Cheng[7,4], Yudong Pan[2,3], Yiqi Liu[2,3], Mengdi Wang[2], Shengwen Liang[1], Yinhe Han[2], Huawei Li[1,3], Xiaowei Li[1,3]

SKLP, Institute of Computing Technology, Chinese Academy of Sciences, Beijing, China[1]

CICS, Institute of Computing Technology, Chinese Academy of Sciences, Beijing, China[2]

University of Chinese Academy of Sciences[3], Nanjing University[4], Shanghai University[5], Zhejiang University[6]

Shanghai Innovation Center for Processor Technologies[7], FuDan University[8]

Hangzhou Institute for Advanced Study, University of Chinese Academy of Sciences[9]

University of Illinois at Urbana Champaign[10], Shanghai Jiao Tong University[11]

Zhejiang Lab[12]

changkaiyan@live.com, wangying2009@ict.ac.cn *

Abstract

Recent advances in large language models have demonstrated their potential for automated generation of hardware description language (HDL) code from high-level prompts. Researchers have utilized fine-tuning to enhance the ability of these large language models (LLMs) in the field of Chip Design. However, the lack of Verilog data hinders further improvement in the quality of Verilog generation by LLMs. Additionally, the absence of a Verilog and electronic design automation (EDA) script data augmentation framework significantly increases the time required to prepare the training dataset for LLM trainers. This paper proposes an automated design-data augmentation framework, which generates high-volume and high-quality natural language aligned with Verilog and EDA scripts. For Verilog generation, it translates Verilog files to an abstract syntax tree and then maps nodes to natural language with a predefined template. For Verilog repair, it uses predefined rules to generate the wrong verilog file and then pairs EDA Tool feedback with the right and wrong verilog file. For EDA Script generation, it uses existing LLM(GPT-3.5) to obtain the description of the Script. To evaluate the effectiveness of our data augmentation method, we finetune Llama2-13B and Llama2-7B models using the dataset generated by our augmentation framework. The results demonstrate a significant improvement in the Verilog generation tasks with LLMs. Moreover, the accuracy of Verilog generation surpasses that of the current state-of-the-art open-source Verilog generation model, increasing from 58.8% to 70.6% with the same benchmark. Our 13B model (ChipGPT-FT[1]) has a pass rate improvement compared with GPT-3.5 in Verilog generation and outperforms in EDA script (*i.e.,* SiliconCompiler) generation with only 200 EDA script data.

Keywords

Data Augmentation, Hardware Generation Large Language Model

1 Introduction

Recent advances in large language models (LLMs) have demonstrated great potential for automated generation of hardware description language (HDL) code from high-level prompts[2–4] and EDA flow coordination [5]. It sheds light on the approach to agile chip development through natural language hardware design, where designers articulate requirements with prompts. Such a change could potentially revolutionize chip design by maximizing designer creativity and efficiency at scale.

Although general LLMs like GPTs can be adapted to a wide range of fields[6, 7], finetuning open-source LLMs is necessary to enhance the capability of democratized chip design process through domain customization, as it does in *e.g.,* NVIDIA ChipNemo[8] and academic community[2, 5, 9]. Compared to prior finetuned LLMs on Verilog code completion, however, as shown in Fig. 1, we believe a true Chip Design Agent based on LLM should be able to act as an interface between human developers by conducting natural language to Verilog code translation, architecture design, feedback analysis from EDA tools and Verilog/script remodification, besides Verilog completion as in prior code-gen LLMs. Such an LLM agent works like a human programmer by interacting with EDA tools feedback to remodify the Verilog code and script. Such an ambitious attempt to reach chip-design LLMs requires very high-quality hardware-specific datasets, which are very limited in the open-source community as shown in Fig.2, and the situation poses a significant challenge to the goal of finetuning chip-design LLMs. In general, there are three major obstacles to address before the successful finetuning of a chip-design LLM. First of all, high-volume hardware datasets including Verilog and hardware description are in demand, while the open-source Verilog codes are very scarce regarding the million-scale requirement of domain-specific LLM finetuning. As we can see from Fig. 3, it takes at least 10^8 entries of code and labels to unleash the intelligence of a domain-specific LLM.

Second, in addition to data volume, the quality of hardware datasets is also important in terms of diversity and consistency. For diversity, there are at least three types of datasets that are required such as natural language and HDL description, EDA feedback and scripts. As to consistency, for Verilog codes, a chip-design LLM needs well-aligned pairs of natural language description and Verilog code, which respectively correspond to input and desired output of a LLM. Besides Verilog codes, error feedback from EDA tools and the corresponding information on Verilog/script modification should also be provided and aligned, so that it can understand and manipulate EDA Tool through scripts such as the SiliconCompiler[10] python script.

To address the data scarcity issue that limits the capability of a chip-design LLM, this paper proposes a full-stack hardware design-data augmentation framework to solve the above challenges. First of all, we propose an automated design-data augmentation framework to generate large-scale hardware design data without human intervention. For the Verilog generation task, we proposed a Verilog program analysis method to achieve alignment between Verilog and natural language, which maps the Verilog AST(Abstract Syntax Tree) directly to natural language. For Verilog repair task, we use the feedback from Verilog semantic checker (*i.e.,* yosys) paired with the provided right and wrong Verilog files to generate Verilog repairing dataset , which aligns the generated Verilog to the response of EDA Tool. For EDA Tool script generation, we observe that although existing

*Ying Wang is the corresponding author.

[1]https://github.com/aichipdesign/chipgptft

DAC '24, June 23–27, 2024, San Francisco, CA, USA

ACM ISBN 979-8-4007-0601-1/24/06.

https://doi.org/10.1145/3649329.3657356

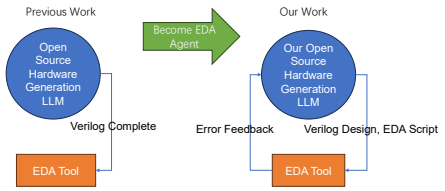

Figure 1: Hardware Generation Large Language Model as an EDA Tool Agent.

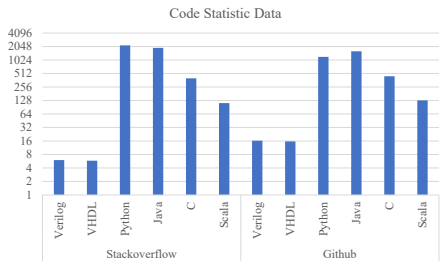

Figure 2: Compare different languages dataset scale. Hardware language dataset is less than software language dataset.

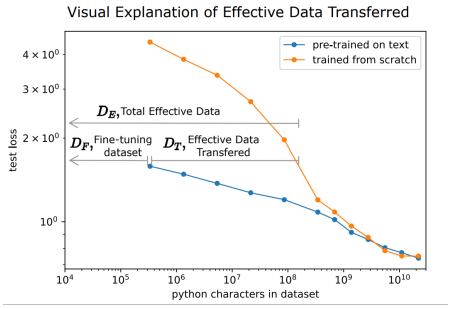

Figure 3: From OpenAI Technique Report for Scaling law in language model[1]. As the dataset size improves, the loss decreases.

LLM (OpenAI GPT) cannot generate accurate EDA Script, it can understand EDA script well. Therefore, we leverage existing LLMs to produce natural language descriptions according to the EDA script. The generated natural language description and EDA script are paired and linked to the same entry of the dataset.

We evaluate our data augmentation framework by finetuning Llama-2 7B and 13B and test it on third-party benchmarks. The results show that the chip design model finetuned with our design-data framework show a higher pass rate over the baseline models(GPT3.5, General Code generation, Thakur et al.[2]) in Verilog generation, Verilog Repair and EDA Script Generation. The contributions are listed below:

- We design and implement a design-data augmentation framework for training Chip Design LLMs that can generate Verilog, EDA script, and coordinate EDA-flow by receiving only natural language design description. The framework facilitates the automatic generation of high-volume and high-quality datasets including aligned Verilog/EDA-script information. To align natural language and Verilog, we use the program analysis method to generate natural language descriptions, which increases the pass rate from 25.7% to 45.7% in 13B model compared with the naive data generation method(*i.e.*, only code completion).
- We use the data generated by the proposed framework to finetune Llama 2-7B, 13B, which improves from 31.4% to 45.7% in function pass rate compared with Thakur et al.[2]. It has a decent pass rate close to GPT-3.5 and outperforms it in some tasks.
- The evaluation also shows that the proposed data augmentation framework maintains the data distribution and enhances the Verilog generation quality over the baseline dataset with the same size. In evaluation, we obtain 3671k data from this framework, which can promote the pass rate from 25.7% to 45.7% when applied to a 13B model compared with the general data generation method(only code completion).

2 Background & Motivation

Chip-design Large Language Model. Large language models (LLMs) have emerged as a promising technique for chip design. Previous research has investigated the utilization of LLMs for the generation of Hardware Description Language (HDL) code, such as Verilog, from natural language descriptions of hardware modules[2, 11, 12]. Progress has been made on tasks like generating complete Verilog code [9], general register-transfer level (RTL) synthesis [13], and enhancing open-source LLMs for EDA tool script writing [5]. LLM-based EDA has also been applied to domains like quantum computing [14], in-memory computing [15], hardware verification [16–18], and AI accelerator design [19]. These efforts demonstrate the great potential of LLMs for automating hardware generation. However, most rely on proprietary commercial models. Developing open-source, portable LLM solutions is crucial for widespread adoption. Tab. 1 summarizes recent methods for finetuning LLMs on hardware generation tasks. For example, NVIDIA finetuned a model called ChipNeMo using proprietary data, precluding full reproducibility [8]. We aim to enable high-quality finetuning of open-source LLMs without reliance on private data or specialized hardware.

Table 1: Comparison of hardware generation large language models.

Works	Target Task	Pre-Trained Model	Target Language	Data	Auto Aug.
ChipNeMo[8]	Verilog Generation	Llama 2	Verilog	Private	×
Thakur et al.[2]	Verilog Completion	CodeGen	Verilog	Github etc.	×
ChatEDA[5]	EDA Script Generation	Llama 2	ChatEDA (Python DSL)	Custom	×
Ours	Verilog Generation, Verilog Repair, EDA Script Generation	Llama 2	Verilog, SiliconCompiler (Python DSL)	Github etc.	✓

Hardware Data Scarcity & Large Language Model is Data-limited under some circumstance. As deep learning models have transitioned to the large language model era, the primary bottleneck has shifted from model architecture to available data[20]. OpenAI demonstrated a "scaling law" where increasing model size alone plateaus, and performance gains depend on more data as shown in Fig. 3. Data augmentation has thus become a key technique for improving models' generalization through exposure to a richer, more diverse training distribution[21]. In hardware domains like Verilog, limited labeled data presents unique challenges for applying large language models (LLMs). As Fig. 2 shows, Verilog code repositories contain orders of magnitude fewer files than general languages. While sufficient for basic code completion, the scarcity of paired natural language descriptions prevents direct training for translation tasks. Additionally, lacking syntax analysis from Verilog checkers precludes LLMs from learning automatic error correction abilities. EDA script generation faces similar data scarcity issues. For some applications, creating extensive labeled training datasets is the most labor-intensive process in machine learning development[5]. To address these hardware domain data limitations, we propose a comprehensive data augmentation framework. It can overcome the following challenges in previous works.

Challenge 1: Natural language Verilog generation requires strict alignment between Verilog and Natural language. During the LLM finetuning process, the generated Verilog should be aligned with the input natural language description. However, in previous LLM finetuning

process, only Verilog file finetune without any natural language comment may cause pass rate decrease as shown in Tab. 5. Therefore, we use a Verilog program analysis to help natural language and Verilog alignment, which can generate corresponding natural language from Verilog file. The core part is natural language generation using Antlr4, which is a parser. Our framework provides Verilog file and grammar file then we can do program analysis on the abstract syntax tree. The program analysis stage uses Verilog Abstract Syntax Tree to compile each syntax node to natural language, which can obtain alignment natural language only with Verilog file.

Challenge 2: Verilog repair requires strict alignment between Verilog and EDA Tool Error feedback. Previous Chip-design LLM[2] can not leverage EDA Tool feedback to correct wrong Verilog. To solve this problem, we use rule-based approach to generate error Verilog file from right Verilog file and obtain the Verilog checker's(Yosys) output. This method can generate Verilog repair with EDA tool alignment from Verilog file.

Challenge 3: EDA script generation requires high alignment with EDA script and Natural language. Previous EDA script generation only uses script as the input of LLM[5]. However, for the lack of EDA Tool example script and the natural language description, LLM is hard to setup connections between natural language and EDA script, which nearly output wrong file. To solve this problem, we observe that existing LLM(Pre-trained LLM, *e.g.* OpenAI GPT) can understand EDA script but can not generate EDA script. Therefore, we give the existing LLM right EDA script and get the corresponding natural language, which obtains a high quality dataset and leverages the LLMs' understanding ability.

3 Automated Design-Data Augmentation Framework

To develop an LLM capable of serving as an automated hardware generation EDA tool agent (Fig. 1), our proposed methodology involves the multi-stage data generation workflow depicted in Fig. 4. The training data generated through this process contains three key fields: 1) An instruction field that distinguishes the expected task (e.g.,code generation vs. error checking). 2) An input field containing the contextual information or prompt for the task. 3) An output field with the corresponding expected result.

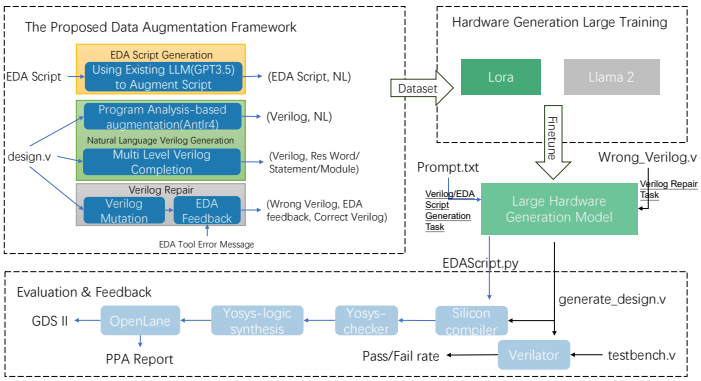

Figure 4: Overall workflow for hardware generation LLMs incorporating the proposed data augmentation framework.

3.1 Data Augmentation for Verilog Generation

To augment Verilog data for tasks beyond basic code completion, we propose a two-stage data augmentation process. It is well understood that LLMs tend to be strongly influenced by recent training examples[22]. Accordingly, our augmentation framework first exposes the model to larger quantities of less refined data to expand its initial knowledge base. This is followed by a second stage involving higher quality, more precisely targeted samples to refine the model's abilities.

3.1.1 Basic Verilog Completion Augmentation Stage Code completion is a widely used data augmentation technique for sequence generation tasks. To strengthen an LLM's ability to predict Verilog code, we employ completion as the basic approach. Verilog completion can be formulated as using an initial token sequence $\{c_0, c_1, \cdots, c_{n-1}\}$ to predict the remaining tokens $\{c_n, c_{n+1}, \cdots, c_m\}$, where c_i represents a character. In our framework, completion examples are represented as: {"instruct":"complete the next [level] of Verilog file.", "input":"[Existing Verilog]","output":"[Predict Verilog]"}. The Verilog prediction data is separated into three coding granularity levels: module-level, sentence-level, and token-level. Module-level prediction uses the module header to generate the body. Sentence-level predicts the next statement given prior code ending in ';'. Token-level forecasts the subsequent token. A Verilog module with i token, j sentence can be divided into $1+j+i$ segments. As our evaluation in Tab.5 indicates, code completion alone yields a small pass rate improvement from 22.9% to 25.7% (Tab. 5) compare with naive Llama 2, necessitating additional techniques to better align natural language and Verilog domains. Later framework stages introduce more sophisticated augmentation for broader capabilities.

3.1.2 Natural language and Verilog Alignment - Using Program Analysis Rule To better align natural language with Verilog semantics, we develop a rule-based program analysis to translate Verilog code into structured natural language descriptions. The Verilog file is first parsed into an abstract syntax tree using ANTLR4. Translation rules are then applied to the syntax tree. For example, the rule module head translate `module x(input a, output reg b);` into "`The Verilog module with name [x] has one input [a] and one output [b]. The output is reg.`". This stage can be formulated as $Description = Rule(Verilog)$. Importantly, the ruleset does not capture full Verilog syntax. This mirrors how programmers typically use only core details when using natural language to generate Verilog. This stage generates training data in the format: D = { "instruct": "give me the Verilog module of this description.", "input":"[natural language]", "output":"[Verilog file]" }. Given a Verilog file with k translatable syntax structures, the dataset size increases linearly at $O(k)$. This alignment stage can improve the LLM's pass rate from 25.7% to 45.7% (Tab. 5), which are similar with GPT-3.5 only with 13B weights. The rule-based approach effectively bridges the semantic gap between the natural language and the Verilog semantic.

To illustrate this technique in more detail, we conduct a case study using a sample Verilog module. As shown in Fig. 5, the module is first parsed into an abstract syntax tree using ANTLR. The example then apply rules to extract semantic information from the syntax structure. For example:

- module & port declaration: Compile modules' declarations into natural language. For example, "module counter(clk, rst, en, count)" is compiled to "module <counter> has <four> parts, their names are <clk, rst, en and count>".
- always block declaration: Compile always block declaration into natural language. For example, in Fig. 5, the always block is tranlated into "The sensitive list in <first> trigger block is <on the positive edge> of <clk>.".
- variable declaration: Compile variable declaration into natural language. For example, in Fig. 5, data augmentation framework outputs "<Output> signal <count> has <2>-bit width in range <1:0>. It is a <reg> variable".

3.2 Data Augmentation for Verilog Repair

3.2.1 Basic Verilog Code Repair Augmentation When editing Verilog code, programmers may unintentionally introduce syntactic errors. A "Verilog repair" task aims to automatically correct such incorrect programs. However, training data for this task is challenging to obtain at scale as it requires incorporating realistic mistakes. We address this data scarcity issue through a rule-based targeted Verilog code masking approach. To construct samples mimicking errors, our framework programmatically

Source Code	Natural Language Description	Core Apply Analysis Rules
1 module counter (clk, rst, en, count); 2 input clk, rst, en; 3 output reg [1:0] count; 4 always @(posedge clk) 5 if (rst) 6 count <= 2'd0; 7 else if (en) 8 count <= count + 2'd1; 9 endmodule	Line 1: module <counter> has <four> ports, their names are <clk, rst, en and count>. Line 2: In the <four> ports, <clk, rst, en> are inputs. <clk> has <1>-bit width, <rst> has <1>-bit width, <en> has <1>-bit width. Line 3: <Output> signal <count> has <2>-bit width in range <1:0>. It is a <reg> variable. Line 4: This module has <one> trigger block. Line 4: The sensitive list in <first> trigger block is <on the positive edge> of <clk>. Line 5-8: In this <always> block, <if> <rst> is 1, then initialize <count> to <2'd0>, else if <en> is true, then <add> <2'd1> to the count.	module_declaration : module_keyword module_identifier module_parameter_port_list? list_of_port_declarations? ';' module_item* 'endmodule' ; list_of_port_declarations : '(' port_declaration (',' port_declaration)* ')' | '(' port (',' port)+ ')' | '(' port_implicit ')' | '(' port_explicit ')' | '(' ')' ; module_item : attribute_instance* module_declaration | attri_inst* module_instant | attri_inst* initial_construct | attri_inst* always_construct; module_declaration : net_declaration | reg_declaration | integer_declaration

Figure 5: Natural Language Generation Using Program Analysis Rule. Core parts EBNF program analysis rules.

masks tokens in correct Verilog programs. These tokens correspond to nodes in the Verilog syntax tree parsed by ANTLR4. In this task, the data augmentation framework generates D = { "instruct": "give me correct Verilog according to the given wrong Verilog.", "input":"[wrong Verilog file]", "output":"[right Verilog file]" }. For a Verilog file containing x tokens, this approach yields $2x$ tokens worth of input-output pairs.

To maintain high data quality, the number of changes made to any individual right Verilog module was limited to below five. A case is shown in Fig. 6, our algorithm changes the third column Verilog to the first column Verilog. The following lists key rules that were used to introduce targeted errors into correct Verilog code during the basic code repair data augmentation process:

- Generate Word Missing: Remove keywords, semicolon and operand.
- Generate Type Error: Change `wire` to `reg` or reverse on the abstract syntax tree.
- Generate Width Error: Add or sub the width value in `wire` and `reg` definitions.
- Generate Additional Word Error: Random add non-sense words in Verilog module.
- Generate Logic Error: Random remove logic condition statements in if statement.

3.2.2 Verilog Code Repair Augmentation with EDA Tool Feedback.

When generating Verilog code from natural language, syntactic or semantic errors can occur in the output. We observed that EDA tool feedback provides valuable information to improve code quality. During logic synthesis, Yosys (an ASIC logic synthesizer) checks Verilog syntax correctness and reports any errors. We leverage this capability by running syntax checks on the masked Verilog programs generated for "Verilog repair" task (Sec. 3.2.1). Specifically, each masked program in Sec. 3.2.1 is fed to Yosys to obtain error information. Our framework then pairs this feedback with the original incorrect code sample. Concretely, as depicted in Fig. 6, the data takes the form: D = { "instruct": "give me correct Verilog according to the given wrong Verilog.", "input":"[yosys info],[wrong Verilog file]", "output":"[right Verilog file]" }. By incorporating EDA Tool diagnostics in this manner, our approach grounds Verilog generation in realistic tool constraints.

3.3 Data Augmentation for EDA Tool Script Generation

We selected the SiliconCompiler library as our baseline EDA tool, as it is a widely used open-source Python framework. However, the volume of existing sample scripts within SiliconCompiler's documentation is insufficient for largescale model finetuning. To address this, we propose using LLMs to augment the dataset while ensuring outputs adhere to the constraints

Input Verilog	Input Feedback	Output Verilog
module LFSR_3bit (input [2:0] SW, input [1:0] KEY, output reg [2:0] LEDR); always @(posedge KEY0]) LEDR <= KEY[1] ? SW : {LEDR[2] ^ LEDR[1], LEDR[0], LEDR[2]}; endmodule	./111_3-bit LFSR.v:7: ERROR: syntax error, unexpected ']'	module LFSR_3bit (input [2:0] SW, input [1:0] KEY, output reg [2:0] LEDR); always @(posedge KEY[0]) LEDR <= KEY[1] ? SW : {LEDR[2] ^ LEDR[1], LEDR[0], LEDR[2]}; endmodule

Figure 6: Our framework generated Verilog repair Data with EDA Tool.

of the SiliconCompiler domain. Although direct LLM generation without domain knowledge risks producing scripts that are syntactically correct but semantically invalid, we observe that the existing LLM can understand SiliconCompiler's script and output a right natural language description. Therefore, we provided existing LLM(*i.e.,* GPT-3.5) with around 200 examples of valid SiliconCompiler scripts in Python file format to obtain the dataset as shown in Equ. 1.

$$\textit{GeneralLLM}(\text{SiliconCompiler Script}) = \text{Natural language Desc.} \quad (1)$$

Surprisingly, with only these small samples(*i.e.* 200), our approach outperformed baselines like GPT-3.5 and others one manitude which required far more training data. For this stage, data augmentation framework generates training instances in the format: D = { "instruct": "give me SiliconCompiler script.", "input":"[LLM generated description]", "output":"[SiliconCompiler script]" }. This method relies on LLM tokens, where high computational costs limits its applicability for Verilog generation tasks.

4 Implementation

Dataset This paragraph illustrates the dataset scale after implementing the proposed data augmentation method, The augmented dataset scale is detailed in Tab 2. We trim the data that exceeds the maximum token length, as such data may introduce certain issues during model training.

Table 2: Dataset Scale through Data Augmentation Framework.

Task	Output Data Size	Output Data Number
Natural Language Verilog Generation	1784.24MB	124k
Verilog Mask Completion	2145.29MB	107k
Verilog Debug	523.77MB	240k
Verilog Word-Level Completion	21GB	3700k
Verilog Module-Level Completion	693MB	400k
Verilog Statement-Level Completion	2.9GB	2388k
Natural Language EDA Script Generation	301KB	200

Finetuning Large Language Model We employ LoraNet[23] to finetune Llama2[24] on these datasets, based on the Llama-recipes repository. The model comprises 40 hidden layers, maximum position embeddings of 2048, and is trained over a period of approximately 7 days. Additionally, our methodology involves the following steps: Step 1: Data collection from GitHub and HuggingFace datasets. Step 2: Application of our proposed data augmentation framework. Step 3: Fine-tuning of the pre-trained LLama 2 large language model. Step 4: Utilization of the benchmark for evaluating data augmentation performance. Our evaluation results demonstrate the effectiveness of the proposed data augmentation method.

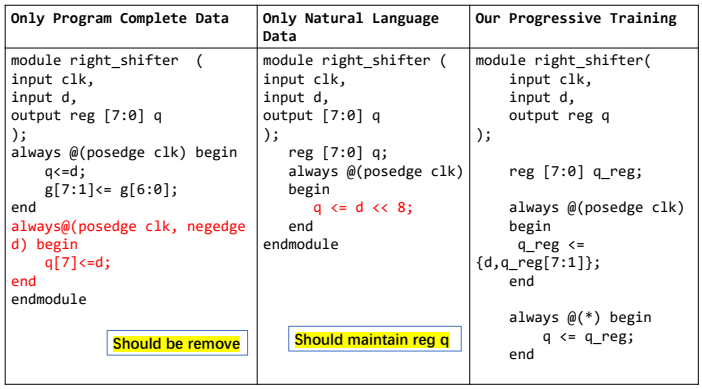

Figure 7: Ablation Study for the Data Augmentation Framework.

4.1 Evaluation Setup

Finetune Environment We harness the extensive language library `transformers` and the distributed framework `deepspeed`, both constructed on the foundational framework PyTorch. Our infrastructure includes a cluster with 8 NVIDIA A100 GPUs, each equipped with 40 GB of memory, and an Intel Xeon CPU.

Baseline We choose Thakur et al.[2, 9] (SOTA Open Source Verilog Generation Model), GPT3.5 as the primary baseline model to access the capabilities of our model. Here are the utilized models for comparsion. ChatEDA[5] is not open-source yet, hence it is excluded from the comparison.

- **Llama 2-PT 13B** LLama 2 is an open-source large language model developed by Meta.
- **Llama 2-FT (Ours) 7B** We employ the proposed data augmentation workflow to fine-tune LLama 2-7B.
- **Llama 2-FT (Ours) 13B** We employ the proposed data augmentation workflow to fine-tune LLama 2-13B.
- **Llama 2-FT (General Aug) 13B** We fine-tune Llama 2-13B only with code completion as an ablation study baseline.
- **GPT 3.5** The most widely used LLM from OpenAI. The weight parameters are not publicly available.
- **Thakur et al.** A Verilog generation large language model based on Codegen-16B.

Benchmark & EDA Environment For the LLM inference, we set temperature to 0.1(less creative) and beams to 4(larger search space). To ensure a fair comparison, we select a subset of Natural language benchmark Thakur et al.[2] and RTLLM[11] as our benchmark. We employ VCS as the functional simulator. The backend of SiliconCompiler operates on openlane, utilizing the SkyWater 130nm Process Design Kit (PDK).

4.2 Evaluation Result

4.2.1 Verilog Generation Comparasion The results in Table 5 indicate that on the Thakur et al. benchmark, our 13B model achieves an 11.8% improvement in pass rate compared to Thakur et al.[2] and a 5.9% improvement compared to GPT-3.5. Additionally, our 13B model improves the pass rate from 25.7% to 45.7% compared to Llama 2-FT (General Aug) 13B, demonstrating the effectiveness of our approach. It shows that our data augmentation framework outperforms the general data generation method on both benchmarks.

4.2.2 Ablation Study To assess the effectiveness of the proposed data augmentation framework, we conduct an ablation study on the Llama2-13B model. We separately finetune LLama 2-13B using a general data generation framework(only code completion) and our proposed data augmentation framework on the same base dataset. The results are shown in Fig. 7. The results indicate that utilizing only code completion framework exhibits flaws in terms of natural language descriptions for Verilog. Besides, in Tab. 5, the success rate of only code completion is 25.7%, which is 45.7% when adding alignment stage. Therefore, we conclude that program analysis alignment takes an important role in fine-tuning large language models for Verilog code generation.

Table 3: Evaluation for Verilog repair using RTLLM[11] benchmark. Syntax represents the number of Verilog code generated by LLM with syntax errors under pass@5. Function represents testbench pass rate of the best-performing Verilog code under pass@5.

Benchmark	ours-13B		ours-7B		GPT3.5		Llama2-13B	
	syntax	function	syntax	function	syntax	function	syntax	function
accu	0	100%	0	100%	0	100%	4	0%
adder_8bit	0	0%	0	0%	1	100%	5	0%
adder_16bit	0	0%	0	0%	2	100%	5	0%
adder_32bit	0	0%	0	0%	4	0%	5	0%
adder_64bit	0	0%	0	0%	5	0%	5	0%
multi_16bit	0	100%	0	100%	5	0%	3	0%
multi_pipe_4bit	0	0%	0	0%	5	0%	1	100%
multi_pipe_8bit	0	100%	3	0%	5	0%	1	0%
multi_booth	0	100%	0	0%	4	0%	0	0%
div_16bit	0	100%	5	0%	4	0%	0	100%
radix2_div	0	100%	0	0%	5	0%	1	0%
Johnson_Counter	0	100%	0	100%	0	97%	3	100%
right_shifter	0	100%	0	100%	0	0%	0	0%
mux	0	100%	0	100%	0	100%	5	0%
counter_12	0	100%	0	100%	4	0%	5	0%
freq_div	0	0%	0	0%	3	0%	0	0%
signal_generator	0	100%	2	0%	5	0%	5	0%
serial2parallel	0	100%	0	100%	3	100%	2	0%
parallel2serial	0	0%	0	0%	5	0%	5	0%
pulse_detect	0	0%	0	0%	2	0%	4	0%
edge_detect	0	100%	0	100%	2	100%	5	0%
fsm	0	100%	0	100%	4	0%	0	0%
width_8to16	0	100%	0	100%	3	30%	4	0%
traffic_light	0	100%	0	100%	5	0%	0	0%
calendar	0	100%	0	100%	2	100%	2	0%
RAM	0	100%	0	100%	1	100%	3	0%
asyn_fifo	0	100%	5	0%	5	0%	2	0%
alu	0	100%	0	100%	5	0%	5	0%
pe	0	100%	0	100%	1	100%	5	0%
success rate		72.40%		51.70%		34.50%		10.30%

Table 4: Evaluation for SiliconCompiler script generation. syn represents the iterations needed to generate SiliconCompiler scripts with correct syntax, while func represents the iterations needed to generate SiliconCompiler scripts with correct function under pass@10.

benchmark	GPT3.5		Thakur et al.[2]		Ours-7B		LLama2-13B		Ours-13B	
	syn.	func.	syn.	func.	syn.	func.	syn.	func.	syn.	func.
Basic	8	9	>10	>10	1	1	>10	>10	1	1
Layout	9	10	>10	>10	1	1	>10	>10	1	1
Clock Period	10	>10	>10	>10	1	1	>10	>10	1	1
Core Area	>10	>10	>10	>10	1	1	>10	>10	1	1
Mixed	>10	>10	>10	>10	2	2	>10	>10	2	2
avg pass@k	>8	>9	>10	>10	1	1	>10	>10	1	1

4.2.3 Verilog Repair Comparasion The benchmark for the Verilog code repair task is derived from syntax-error code generated by the Large Language Model (LLM). As shown in Tab. 3, The performance of our 13B model has been improved by 37.9% compared to GPT-3.5 and by 62.1% compared to the pre-trained Llama2-13B model.

4.2.4 EDA Tool Script Generation Comparasion The benchmark comprises five different levels of EDA script generation tasks, such as the 'core area' case, which represents the setting of the core area in the EDA script. The model takes natural language descriptions as input and produces the corresponding EDA scripts as output. As shown in Tab. 4, The results demonstrate that our 13B model can generate accurate EDA scripts based on Silicon Compiler with just one query, surpassing the performance of models like GPT-3.5 and Thakur et al.[2].

Table 5: Evaluation for Verilog Generation. Every cell in Thakur et al. benchmark[2] consists the result of three prompt levels(low/middle/high). Syntax represents the number of Verilog code generated by LLM with syntax errors under pass@5. Function represents testbench pass rate of the best-performing Verilog code under pass@5.

benchmark		GPT3.5		Ours-7B		Ours-13B		Thakur et al.[2]		Llama2-13B		LLama2-General Aug.	
	Name	syntax	function	syntax	function	syntax	function	syntax	function	syntax	function	syntax	function
Thakur et al.	basic1	0/0/0	100%/100%/100%	0/0/0	100%/100%/100%	0/0/0	100%/100%/100%	0/0/0	100%/100%/100%	0/0/0	100%/100%/100%	0/0/0	100%/100%/100%
	basic2	0/0/0	100%/100%/100%	0/0/0	100%/100%/100%	0/0/0	100%/100%/100%	0/0/0	100%/100%/100%	0/0/0	100%/100%/100%	0/5/0	100%/0%/100%
	basic3	0/0/0	12.5%/37.5%/100%	0/0/0	50%/50%/50%	0/0/0	25%/50%/50%	0/0/0	12.5%/0%/100%	0/0/0	0%/12.5%/100%	0/0/0	12.5%/12.5%/100%
	basic4	0/1/1	100%/100%/100%	0/0/0	100%/100%/100%	0/0/0	100%/100%100%	0/0/0	0%/100%/100%	0/0/0	100%/100%/100%	3/1/0	100%/100%/100%
	intermediate1	0/0/0	100%/100%/100%	0/0/0	25%/25%/25%	2/0/0	25%/100%/100%	0/0/0	100%/100%/0%	0/5/5	100%/0%/0%	0/0/0	75%/75%/100%
	intermediate2	0/0/0	100%/100%/100%	0/0/0	68.4%/68/4%/100%	0/0/0	100%/73.6%/100%	0/0/0	0%/0%100%	0/0/0	0%/0%/100%	0/0/0	0%/0%/100%
	intermediate3	5/5/4	0%/0%/6.25%	5/5/5	0%/0%/0%	1/0/0	11.7%/11.7%/11.7%	0/0/0	0%/0%/6.25%	0/0/0	0%/0%/0%	5/5/5	0%/0%/0%
	intermediate4	0/0/0	0%/0%/0%	0/0/0	50%/62.5%/100%	0/0/0	37.5%/62.5%/75%	1/0/0	0%/0%/0%	5/2/2	0%/0%/0%	0/0/0	0%/12.5%/12.5%
	intermediate5	0/0/0	100%/100%/100%	0/0/0	30%/30%/30%	0/0/0	10%/60%/60%	0/0/0	0%/0%/0%	3/0/0	0%/20%/60%	0/5/5	10%/0%/0%
	intermediate6	0/0/0	0%/0%/0%	5/5/0	0%/0%/100%	5/5/0	0%/0%/100%	0/0/0	0%/0%/0%	5/5/5	0%/0%/0%	5/5/5	0%/0%/0%
	intermediate7	1/1/1	100%/100%/100%	0/0/0	100%/100%/100%	3/5/0	100%/100%/100%	1/0/0	6%/6%/100%	2/2/0	0%/0%/0%	0/1/0	6%/6%/100%
	intermediate8	0/0/0	100%/100%/100%	0/0/5	37.5%/37.5%/0%	0/0/0	25%/80%/80%	2/4/5	0%/0%/0%	0/0/0	62.5%/62.5%/62.5%	0/0/5	0%/0%/0%
	advanced1	3/1/1	100%/100%/100%	0/0/0	49.9%/51.5%/75%	0/0/0	100%/49.9%/51.5%	1/2/5	25%/25%/0%	1/1/0	25.2%/25.2%/0.19%	1/5/0	50%/0%/25%
	advanced2	0/0/0	92.8%/92.8%/92.8%	0/0/0	92.8%/100%/100%	0/0/5	92.8%/100%/100%	0/0/0	92.8%/100%/92.8%	0/5/5	0%/0%/0%	0/0/0	7%/14%/93%
	advanced3	0/0/1	83%/83%/83%	0/0/0	100%/100%/100%	0/0/5	100%/100%/100%	0/0/0	100%/0%/100%	1/0/0	0%/0%/100%	0/0/0	66%/66%/100%
	advanced4	0/0/0	62.5%/62.5%/62.5%	0/0/0	100%/100%/100%	5/5/5	100%/100%/100%	1/0/0	100%/100%/100%	0/0/0	12.5%/12.5%/50%	0/0/0	37.5%/50%/50%
	advanced5	0/0/0	37.5%/37.5%/100%	5/0/0	0%/50%/100%	5/0/0	37.5%/75%/100%	0/0/5	0%/75%/0%	0/1/1	0%/37.5%/0%	0/0/0	37.5%/37.5%/0%
success rate			64.7%		64.7%		70.6%		58.8%		41.2%		47.1%
RTLLM[11]	accu	2	0%	5	0%	5	0%	0	0%	3	0%	5	0%
	adder_8bit	4	0%	4	7%	0	100%	5	0%	3	0%	3	46%
	adder_16bit	1	55%	0	52%	0	70%	3	55%	2	50%	5	0%
	adder_32bit	3	0%	5	0%	0	0%	5	0%	0	0%	0	0%
	adder_64bit	2	0%	5	0%	5	0%	5	0%	5	0%	0	0%
	multi_16bit	0	0%	5	0%	5	0%	2	0%	5	0%	2	0%
	Johnson_Counter	1	97%	5	0%	5	0%	4	97%	5	0%	5	0%
	right_shifter	0	100%	0	0%	0	0%	0	100%	5	0%	0	0%
	mux	1	100%	1	0%	0	100%	5	0%	5	0%	0	0%
	counter_12	4	0%	2	97%	0	93%	2	93%	5	0%	0	50%
	signal_generator	5	0%	5	0%	0	33.30%	0	0%	5	0%	5	0%
	serial2parallel	3	0%	5	0%	5	0%	5	0%	0	0%	0	0%
	edge_detect	0	100%	0	98%	0	96%	1	0%	5	0%	0	100%
	width_8to16	3	30%	2	33%	0	33%	5	0%	0	33%	5	0%
	calendar	2	100%	5	0%	0	100%	5	0%	0	100%	0	0%
	RAM	5	0%	5	0%	5	0%	3	50%	5	0%	5	0%
	alu	5	0%	5	0%	5	0%	5	0%	5	0%	5	0%
	pe	0	100%	0	100%	0	100%	4	0%	0	0%	0	0%
success rate			27.8%		5.6%		22.2%		5.6%		5.6%		5.6%
All success			45.7%		34.3%		45.7%		31.4%		22.9%		25.7%

5 Conclusions

LLM-based chip design has demonstrated substantial promise for automating Verilog and EDA script generation. However, finetuning is currently constrained by the availability of training data. This paper proposed and evaluated a design-data augmentation framework aimed at enhancing the finetuning of LLMs in Verilog code generation domain. Experimental results revealed that the accuracy of Verilog generation surpasses that of the current state-of-the-art open-source Verilog generation model, increasing from 58.8% to 70.6% with the same benchmark and outperforms GPT-3.5 in Verilog repair and EDA Script Generation with only 13B weights.

Acknowledgments

This work was supported by the National Natural Science Foundation of China under NSFC.62090024, 62222411, 62025404, 92373206, 62202453

References

[1] D. Hernandez, J. Kaplan, T. Henighan, and S. McCandlish, "Scaling laws for transfer," *arXiv preprint arXiv:2102.01293*, 2021.

[2] 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)*, pp. 1–6, IEEE, 2023.

[3] 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(MLCAD)*, pp. 27–32, 2020.

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

[5] Z. He, H. Wu, X. Zhang, X. Yao, S. Zheng, H. Zheng, and B. Yu, "Chateda: A large language model powered autonomous agent for eda," *arXiv preprint arXiv:2308.10204*, 2023.

[6] Y. Wei, Z. Wang, Y. Lu, C. Xu, C. Liu, H. Zhao, S. Chen, and Y. Wang, "Editable scene simulation for autonomous driving via collaborative llm-agents," 2024.

[7] B. Jin, X. Liu, Y. Zheng, P. Li, H. Zhao, T. Zhang, Y. Zheng, G. Zhou, and J. Liu, "Adapt: Action-aware driving caption transformer," in *2023 IEEE International Conference on Robotics and Automation (ICRA)*, pp. 7554–7561, 2023.

[8] T. E. Mingjie Liu§, "Chipnemo: Domain-adapted llms for chip design," *arXiv preprint arXiv:2307.09288*, 2023.

[9] 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," *arXiv preprint arXiv:2308.00708*, 2023.

[10] A. Olofsson, W. Ransohoff, and N. Moroze, "A distributed approach to silicon compilation: Invited," in *Proceedings of the 59th ACM/IEEE Design Automation Conference(DAC)*, p. 1343–1346, 2022.

[11] Y. Lu, S. Liu, Q. Zhang, and Z. Xie, "Rtllm: An open-source benchmark for design rtl generation with large language model," in *Asia and South Pacific Design Automation Conference(ASP-DAC)*, 2023.

[12] M. Liu, N. Pinckney, B. Khailany, and H. Ren, "VerilogEval: evaluating large language models for verilog code generation," in *2023 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2023.

[13] J. Blocklove, S. Garg, R. Karri, and H. Pearce, "Chip-chat: Challenges and opportunities in conversational hardware design," *arXiv preprint arXiv:2305.13243*, 2023.

[14] Z. Liang, J. Cheng, R. Yang, H. Ren, Z. Song, D. Wu, X. Qian, T. Li, and Y. Shi, "Unleashing the potential of llms for quantum computing: A study in quantum architecture design," *arXiv preprint arXiv:2307.08191*, 2023.

[15] Z. Yan, Y. Qin, X. S. Hu, and Y. 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.

[16] B. Ahmad, S. Thakur, B. Tan, R. Karri, and H. Pearce, "Fixing hardware security bugs with large language models," *arXiv preprint arXiv:2302.01215*, 2023.

[17] R. Kande, H. Pearce, B. Tan, B. Dolan-Gavitt, S. Thakur, R. Karri, and J. Rajendran, "Llm-assisted generation of hardware assertions," *arXiv preprint arXiv:2306.14027*, 2023.

[18] M. Orenes-Vera, M. Martonosi, and D. Wentzlaff, "From rtl to sva: Llm-assisted generation of formal verification testbenches," 2023.

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

[20] O. H. Hamid, "From model-centric to data-centric ai: A paradigm shift or rather a complementary approach?," in *2022 8th International Conference on Information Technology Trends (ITT)*, pp. 196–199, IEEE, 2022.

[21] S. Yu, T. Wang, and J. Wang, "Data augmentation by program transformation," *Journal of Systems and Software*, vol. 190, p. 111304, 2022.

[22] J. Dodge, G. Ilharco, R. Schwartz, A. Farhadi, H. Hajishirzi, and N. Smith, "Fine-tuning pretrained language models: Weight initializations, data orders, and early stopping," *arXiv preprint arXiv:2002.06305*, 2020.

[23] E. J. Hu, Y. Shen, P. Wallis, Z. Allen-Zhu, Y. Li, S. Wang, L. Wang, and W. Chen, "LoRA: Low-rank adaptation of large language models," in *International Conference on Learning Representations(ICLR)*, 2022.

[24] H. Touvron, L. Martin, K. R. Stone, P. Albert, A. Almahairi, Y. Babaei, N. Bashlykov, S. Batra, P. Bhargava, S. Bhosale, D. M. Bikel, L. Blecher, C. C. Ferrer, M. Chen, G. Cucurull, D. Esiobu, J. Fernandes, J. Fu, W. Fu, B. Fuller, C. Gao, V. Goswami, N. Goyal, A. S. Hartshorn, S. Hosseini, R. Hou, H. Inan, M. Kardas, V. Kerkez, M. Khabsa, I. M. Kloumann, A. V. Korenev, P. S. Koura, M.-A. Lachaux, T. Lavril, J. Lee, D. Liskovich, Y. Lu, Y. Mao, X. Martinet, T. Mihaylov, P. Mishra, I. Molybog, Y. Nie, A. Poulton, J. Reizenstein, R. Rungta, K. Saladi, A. Schelten, R. Silva, E. M. Smith, R. Subramanian, X. Tan, B. Tang, R. Taylor, A. Williams, J. X. Kuan, P. Xu, Z. Yan, I. Zarov, Y. Zhang, A. Fan, M. Kambadur, S. Narang, A. Rodriguez, R. Stojnic, S. Edunov, and T. Scialom, "Llama 2: Open foundation and fine-tuned chat models," *ArXiv*, vol. abs/2307.09288, 2023.