2024 IEEE LLM Aided Design Workshop (LAD) | 979-8-3503-7608-1/24/$31.00 ©2024 IEEE | DOI: 10.1109/LAD62341.2024.10691788

RTLCoder: Outperforming GPT-3.5 in Design RTL Generation with Our Open-Source Dataset and Lightweight Solution

Shang Liu
HKUST, sliudx@connect.ust.hk

Wenji Fang
HKUST, wenjifang1@ust.hk

Yao Lu
HKUST, yludf@connect.ust.hk

Qijun Zhang
HKUST, qzhangcs@connect.ust.hk

Hongce Zhang
HKUST (GZ) & HKUST, hongcezh@ust.hk

Zhiyao Xie*
HKUST, eezhiyao@ust.hk

***Abstract*—The automatic generation of RTL code (e.g., Verilog) using natural language instructions and large language models (LLMs) has attracted significant research interest recently. However, most existing approaches heavily rely on commercial LLMs such as ChatGPT, while open-source LLMs tailored for this specific design generation task exhibit notably inferior performance. The absence of high-quality open-source solutions restricts the flexibility and data privacy of this emerging technique. In this study, we present a new customized LLM solution with a modest parameter count of only 7B, achieving better performance than GPT-3.5 on all representative benchmarks for RTL code generation. Especially, it outperforms GPT-4 in VerilogEval Machine benchmark. This remarkable balance between accuracy and efficiency is made possible by leveraging our new RTL code dataset and a customized LLM algorithm, both of which have been made fully open-source[1].**

I. Introduction

In recent years, large language models (LLMs) such as GPT [1] have demonstrated remarkable performance in natural language processing (NLP). Inspired by this progress, researchers have also started exploring the adoption of LLMs in agile hardware design. Many new LLM-based techniques emerge and attract wide attention in 2023. For example, LLM-based solutions are proposed to generate design flow scripts to control EDA tools [2], [3], design AI accelerator architectures [4], [5], design quantum architectures [6], hardware security assertion generation [7], fix security bugs [8], and even directly generate the target design RTL [3], [9]–[15].

Among the above explorations, a promising direction that perhaps attracts the most attention is automatically generating design RTL based on natural language instructions [3], [9]–[15]. Specifically, given design functionality descriptions in natural language, LLM can directly generate corresponding hardware description language (HDL) code such as Verilog, VHDL, and Chisel from scratch. Compared with well-explored *predictive* machine learning (ML)-based solutions in EDA [16], such *generative* methods benefit the hardware design and optimization process more directly. This LLM-based design generation technique can potentially revolutionize the existing HDL-based VLSI design process, relieving designers from the tedious HDL coding tasks.

Table I summarizes existing work on LLM-based design RTL generation. Some works [9]–[11], [14], [15] focus on prompt engineering methods based on commercial LLMs like GPT, without proposing new datasets or models for RTL code generation. As we will discuss later, reliance on commercial LLM tools limits in-depth research exploration and incurs serious privacy concerns in industrial IC design scenarios. Thakur et al. [13] generate a large unsupervised training[2] dataset by collecting Verilog-based projects from online resources like GitHub, then fine-tuning its own model. However, this unsupervised dataset is quite unorganized with a mixture of code and text. Evaluations on a third-party benchmark [11] show that the performance of its fine-tuned model is still inferior to commercial tools like GPT-3.5. The VerilogEval [12] from the NVIDIA research team proposes its own labeled training dataset and benchmark, then fine-tunes its own new model. This may be the first non-commercial model that claims comparable performance with GPT-3.5, but according to their authors, neither the training dataset nor fine-tuned LLM model will be released to the public in the near future [12]. Besides these customized RTL-generation solutions, according to our study, all other software code (e.g., Python) generation models like CodeGen2 [19], StarCoder [20], and Mistral [21] are significantly inferior to GPT-3.5 in this RTL generation task.

Works	New Training Dataset	New LLM Model	Outperform GPT-3.5
Prompt Engineering [9]–[11], [14], [15]	N/A	N/A	N/A
Thakur et al. [13]	**Open-Source**	**Open-Source**	No
VerilogEval [12] ChipNeMo [3] ChipGPT-FT [17]	Closed-Source	Closed-Source	Comparable
BetterV [18]			**Yes**
RTLCoder	**Open-Source**	**Open-Source**	**Yes**

TABLE I: LLM-based works on automatic design RTL (e.g., Verilog) generation based on natural language instructions.

Compared with solutions based on closed-source commercial LLM tools like GPT, the open-source LLM solution is vitally important from both research and application perspectives: 1) For research purposes, obviously, closed-source commercial tools prevent most in-depth studies and customizations of this emerging technique. 2) For realistic applications, users of commercial LLM tools unavoidably have data privacy concerns,

*Corresponding Author

[1]RTLCoder has been open-source in https://github.com/hkust-zhiyao/RTL-Coder. It includes the data generation flow, the complete generated training data set, the model training flow, and the final fine-tuned models (based on both Mistral and DeepSeek).

[2]Most customized LLM solutions (including RTLCoder) are developed by fine-tuning pre-trained LLMs based on a training dataset about the specific task. In this paper, we use the terms *training* and *fine-tuning* interchangeably.

since all instructions have to be uploaded to LLM providers like OpenAI. In comparison, each user's own local LLM developed based on an open-source solution can eliminate all privacy concerns and also ensure a reliable service.

However, as mentioned, high-performance open-source RTL generation models are currently unavailable. According to our study, a major challenge is the unavailability of high-quality circuit design data for training: 1) Organized design data is mostly owned by semiconductor companies, who are almost always unwilling to share design data. 2) Design data directly collected online is messy and unorganized, either leading to inferior model performance or requiring prohibitive human efforts to clean the dataset.

In this work, we finally fill this gap with our new open-source LLM solution named **RTLCoder**. To the best of our knowledge, it is the first open-source LLM that outperforms GPT-3.5 in all representative RTL code generation benchmarks [11], [12]. Our contributions are summarized below.

- Targeting Verilog code generation, we propose an automated flow to generate a large labeled dataset with over 27 thousand diverse Verilog design problems and answers. It addresses the serious data availability challenge in IC design-related tasks, and its potential applications are not limited to LLMs. LLM directly trained on it can already achieve comparable accuracy to GPT-3.5.
- We introduce a new memory-efficient LLM training scheme based on code quality feedback. It further boosts the ultimate model performance to outperform GPT-3.5, being comparable to GPT-4. Our 7B model can be trained with only four commercial GPU cards.
- RTLCoder has been fully open-sourced, including our data generation flow, complete generated dataset, LLM training algorithm, and the fine-tuned model. Considering RTLCoder's lightweight property and low hardware barrier, it allows anyone to easily replicate and further improve based on our existing solution.

II. Automatic Dateset Generation

In this work, we first propose a new automated training dataset generation flow. Based on this flow, we have generated over 27 thousand training samples, with each sample being a pair of design instruction (i.e., model input) and the reference RTL code (i.e., expected model output). The instruction can be viewed as the input question for LLMs, describing the desired circuit functionality in natural language. The reference code is the expected answer from LLMs, implementing the circuit functionality in Verilog code. We observe that these generated training samples exhibit high diversity and complexity in the RTL-generation domain, encompassing a diverse spectrum of difficulty levels.

We build this automated generation flow by taking full advantage of the powerful general text generation ability of the commercial tool GPT. Please notice that GPT is only used for dataset generation in this work and GPT-3.5 is adopted here. The automated dataset generation flow is illustrated in Figure 1, which includes three stages: 1) RTL domain keywords preparation, 2) instruction generation, and 3) reference code generation. We designed several general prompt templates to control GPT generating the desired outputs in each stage.

A. Stage 1: Keywords Preparation

The first stage of our data generation flow targets preparing RTL domain keywords for subsequent stages. At process ❶ shown in Figure 1, we request GPT to generate keywords related to digital IC design (i.e., commonly used logic components) based on a set of prompts P_{key}. We obtain a keyword pool $\mathcal{L}_{key}$ with hundreds of digital design keywords.

Specifically, in this process ❶, to collect a comprehensive range of RTL design task topics, we utilize a tree-like structure with multiple branches to issue queries to GPT. We first prompt GPT at the root node to provide categories and examples of frequently used block keywords in RTL design. The response from GPT has a tree structure that consists of some related subfields. With the response, we could use the categories and examples as branches to continue prompting GPT for more design keywords within each topic. For example, we can use scripts to ask GPT about more types of the block "multiplier", it will return more specific design names such as "Booth multiplier, Wallace tree multiplier, etc.". After this process, we obtain hundreds of keywords related to RTL design in the Keywords pool $\mathcal{L}_{key}$.

B. Stage 2: Instruction Generation

The second stage targets generating sufficient instructions based on the initial keywords and Verilog source code. At process ❷, we extend existing keywords from $\mathcal{L}_{key}$ to complete instructions. Specifically, we randomly sample one or two keywords from $\mathcal{L}_{key}$ each time, combined with prompts P_{ext}, and feed them into GPT to obtain an RTL design instruction.

In addition to keyword-based instruction generation in process ❷, we also propose to generate instructions based on existing source code collected by us, as shown in process ❸. This is partially inspired by the work of [22]. By providing GPT with either part or a complete Verilog code $\mathcal{L}_{code}$ collected by [13], we can inspire it to create a related Verilog design problem. By adopting this new ❸ together with ❷, we further enhance the diversity of our dataset by utilizing a vast and varied collection of source code.

Process ❷ and ❸ help generate the initial design instruction pool $\mathcal{L}_{ins}$ based on our customized prompt P_{ext}. After generating the initial instruction pool $\mathcal{L}_{ins}$ with hundreds of initial instructions, we will iteratively use mutation methods to significantly augment the scale and complexity of this pool. At ❹, we use P_{mut} to apply two types of mutation operations on instructions sampled from the design instruction library $\mathcal{L}_{ins}$. The process ❺ would check every new design instruction using a set of rules and only passed valid instructions are added to $\mathcal{L}_{ins}$. Stage 2 is fully automated and accurate enough to generate a high-quality ultimate instruction pool $\mathcal{L}_{ins}$, including over 50,000 instructions.

In addition, we will further request GPT to generate its reasoning steps (i.e., how it analyzes the generation task step-by-step). These reasoning steps further enhance the detailed information of our instruction pool.

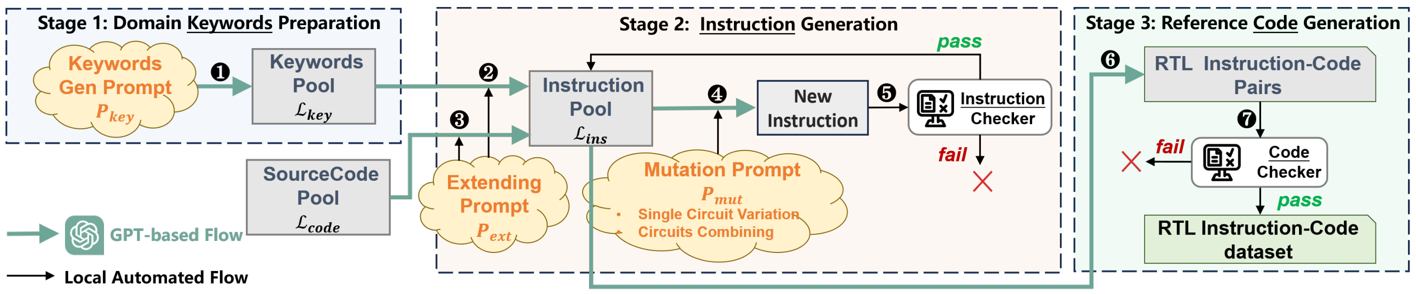
Stage 1: Domain Keywords Preparation
Keywords Gen Prompt P_{key}
❶
Keywords Pool $\mathcal{L}_{key}$
SourceCode Pool $\mathcal{L}_{code}$
GPT-based Flow
Local Automated Flow
Stage 2: Instruction Generation
❷
❸
Instruction Pool $\mathcal{L}_{ins}$
Extending Prompt P_{ext}
❹
New Instruction
❺
Instruction Checker
pass
fail
Mutation Prompt P_{mut}
Single Circuit Variation
Circuits Combining
Stage 3: Reference Code Generation
❻
RTL Instruction-Code Pairs
❼
Code Checker
fail
pass
RTL Instruction-Code dataset

Fig. 1: Our proposed automated training dataset generation flow.

C. Stage 3: Reference Code Generation

The third stage targets generating the reference code. In the third stage, as shown in ❻, we feed each instruction from $\mathcal{L}_{ins}$ into GPT, generating 5 corresponding reference design codes as the solution candidates. After that, in ❼, we will evaluate these answers using a code checker. In this work, we adopt an automated syntax checker and only syntax-correct design code can be kept. If all 5 answers fail the syntax checking, this instruction will be discarded. Finally, only valid instruction-code pairs are saved as our dataset. Ideally, process ❼ should also check whether the functionality of the generated RTL code is consistent with the instruction, but currently generating testbenches for functionality verification cannot be automated. This imperfect automated checking can already filter out the most serious mistakes in the dataset.

After going through all three stages, we generate the ultimate training dataset with more than 27,000 data samples. An interesting observation is that, although we generate our training dataset based on GPT-3.5, RTLCoder turns out to outperform GPT-3.5 on representative benchmarks [11], [12]. One important reason is that, for each instruction, we have employed a syntax checker to filter out the obviously incorrect codes generated from GPT-3.5 and retain the largely correct ones for training RTLCoder. This process can be viewed as a refinement of GPT-3.5's Verilog generation capabilities.

III. New Training Scheme Incorporating Code Quality Feedback

The sequence generation is autoregressive, which means the model always predicts the next token based on its own generated previous ones rather than the reference tokens. Therefore, the traditional model tuning based on maximum likelihood estimation (MLE) would result in a phenomenon named *exposure bias* [23], [24] and the trained model would still generate many low-quality codes. To alleviate this phenomenon, we propose a new LLM training scheme that incorporates code quality scoring. It further improves the RTLCoder's performance on the RTL generation task.

For each instruction, we will now collect multiple additional code candidates generated by the initial pre-trained model. Then we pack these candidates and the original reference code y_i together as $\mathbf{y}_i = \{y_{i,k}\}$, $k = 1, 2, .., K$, where K represents the number of generated code for one instruction x_i. Next, all these candidates will be scored by the scoring mechanism $R(x_i, y_{i,k})$ which could be a syntax checker or unit test for functionality check. We will then obtain a set of score $\mathbf{z}_i = \{z_{i,k}\}$, $k = 1, 2, .., K$, denoting the quality for the code sample $\{y_{i,k}\}$. In the training process, we make the model learn to assign relatively higher generation probabilities to answers with higher scores.

To further make this training scheme more memory efficient, we decompose the computation graph calculation and use the gradient accumulation-alike method to reduce the space complexity from $O(K)$ to $O(1)$.

IV. Experimental Results

A. Evaluation Benchmark and Metric

To evaluate the performance of Verilog code generation, there are two representative benchmarks VerilogEval [12] and RTLLM [11]. The VerilogEval [12] benchmark consists of two parts, EvalMachine and EvalHuman, each including more than 100 RTL design tasks. We follow the original paper [12] and use the widely-adopted $pass@k$ metric. The RTLLM V1.1 [11] benchmark contains 29 RTL design tasks at a larger design scale. We use Synopsys VCS [25] to calculate the scores of the design syntax part and design functionality part separately. In both parts, following the original benchmark [11], each task is counted as success as long as *any* of 5 trials passes the test. This can be interpreted as pass@5 metric. In the generation process, we set $top_p = 0.95$ and $temperature = \{0.2, 0.5, 0.8\}$. For all tested models (i.e., baselines, RTLCoder, and ablation studies), we evaluate all 3 $temperature$ conditions and report the best of each model.

B. Model Training

To ensure a fair evaluation of our proposed RTLCoder, before training, we explicitly examined the similarity between samples in our proposed training dataset and those test cases in benchmarks [11], [12] using Rouge-L metric. Then we get rid of our training samples that are highly similar to test cases during the training process.

Based on our generated dataset with 27K instruction-code pairs, we choose the latest Mistral-7B-v0.1 [21] and DeepSeek-Coder-6.7b [26] as the basic pre-trained model for finetuning. In all experiments, we opted for the Adam optimizer with β_1 = 0.9, β_2 = 0.999, and learning rate γ = 1e-5, while abstaining from the use of weight decay. Concurrently, we established a context length of 2048 and a global batch size of 256. We trained the model on only 4 consumer-level RTX 4090 GPUs (24GB each),

Model Type	Evaluated Model	Num of Params	VerilogEval Benchmark [12] (using pass@k metric)						RTLLM V1.1 [11] (using pass@5 metric)	
			Eval-Machine (%)			Eval-Human (%)			Syntax-VCS	Func
			k=1	k=5	k=10	k=1	k=5	k=10	(%)	(%)
Closed-Source Baseline	GPT-3.5	N/A	46.7	69.1	74.1	26.7	45.8	51.7	89.7	37.9
	GPT4	N/A	60.0	70.6	73.5	43.5	55.8	58.9	100	65.5
	ChipNeMo* [3]	13B	43.4	N/A	N/A	22.4	N/A	N/A	N/A	N/A
	VerilogEval* [12]	16B	46.2	67.3	73.7	28.8	45.9	52.3	N/A	N/A
	BetterV* [18]	7B	64.2	75.4	79.1	40.9	50.0	53.3	N/A	N/A
Open-Source Baseline	Codegen2 [19]	16B	5.00	9.00	13.9	0.90	4.10	7.25	72.4	6.90
	Starcoder [20]	15B	46.8	54.5	59.6	18.1	26.1	30.4	93.1	27.6
	Thakur et al. [13]	16B	44.0	52.6	59.2	30.3	43.9	49.6	86.2	24.1
Base Model	Mistral-7B-v0.1 [21]	7B	36.9	48.8	57.4	4.49	12.6	18.6	72.4	20.7
	DeepSeek-Coder-6.7b [26]	6.7B	54.1	63.8	67.5	30.2	42.2	46.2	89.6	34.5
Less Training Data (10K Samples)	RTLCoder-Mistral-10k	7B	56.5	66.6	69.4	31.7	42.2	46.5	86.2	34.5
	RTLCoder-DeepSeek-10k	6.7B	55.3	70.4	76.2	36.7	47.0	50.4	79.3	37.9
Direct Training	RTLCoder-Mistral-Direct	7B	58.9	70.0	74.1	34.4	42.3	45.1	89.7	41.4
	RTLCoder-DeepSeek-Direct	6.7B	59.8	73.6	77.2	39.1	48.3	51.3	86.2	44.8
RTLCoder	**RTLCoder-Mistral**	7B	62.5	72.2	76.6	36.7	45.5	49.2	96.6	48.3
	RTLCoder-DeepSeek	6.7B	61.2	76.5	81.8	41.6	50.1	53.4	93.1	48.3

*We cannot directly evaluate VerilogEval [12], ChipNeMo [3] and BetterV [18] on RTLLM Benchmark due to closed-source models. We fully understand and respect the authors' privacy concerns. The accuracy values of VerilogEval [12], ChipNeMo [3], BetterV [18], GPT-3.5, and GPT-4 on the VerilogEval Benchmark [12] are directly cited from the original publication [3], [12], [18].

TABLE II: Performance comparison of RTL code generators on VerilogEval Benchmark [12] and RTLLM Benchmark [11]. The top scores ranked 1st, 2nd, and 3rd in each column are marked in Green, Blue, and Red, respectively. RTLCoder outperforms GPT-4 on EvalMachine of [12]. It is only second to GPT-4 on the other benchmarks.

each of which could only afford 2×2048 context length using DeepSpeed stage-2 [27].

To implement our proposed training scheme, we first generated 3 code candidates for each instruction using a pre-trained model with Beam search method. Then we use Pyverilog [28] as the syntax checker to score the code candidates.

C. *Experiment Results Overview*

Table II summarizes the comparison of all relevant RTL generation solutions, including commercial models GPT3.5/GPT4, models customized for Verilog generation [12], [13] [18], software code generators [19]–[21], and ablation studies of RTLCoder.

In the VerilogEval benchmark [12], for both EvalMachine and EvalHuman categories, RTLCoder-DeepSeek scores 61.2 and 41.6 respectively. It clearly outperforms GPT-3.5 and is only inferior to GPT-4 among all the models in EvalHuman. Specifically, in the EvalMachine part, RTLCoder-DeepSeek and RTLCoder-Mistral even outperform GPT4 by an absolute value of 1.2% and 2.5%. A similar trend can be observed in the RTLLM benchmark V1.1 [11]. RTLCoder is second only to GPT-4. In summary, RTLCoder outperforms GPT-3.5 and all non-commercial baseline models in most of the metrics.

Furthermore, we validate the effectiveness of our proposed dataset and algorithm through an ablation study. The RTLCoder-Mistral-Direct and RTLCoder-DeepSeek-Direct are directly trained with the traditional MLE method. Using our training dataset, they can already significantly outperform the base model and even GPT-3.5 on part of these indexes. Then the RTLCoders trained with our proposed training scheme further outperform those using Direct training method on all benchmarks, indicating that our training method greatly further improves the model performance.

We also randomly selected 10K samples from the 27K training dataset to finetune the base models and obtained RTLCoder-Mistral-10k and RTLCoder-DeepSeek-10k respectively. Compared with the two models, RTLCoders trained on a 27K dataset are clearly superior on all metrics. Increasing the size of the training dataset and enhancing its diversity clearly further improves the model performance.

V. Conclusion

This work presents a fully open-sourced LLM solution named RTLCoder for RTL code generation, achieving state-of-the-art performance in non-commercial solutions and outperforming GPT-3.5. We contribute a new data generation flow and a complete dataset with over 27 thousand labeled samples, addressing the serious data availability problem in hardware-design-related tasks. Also, we contribute a new training scheme based on design quality scoring. It greatly boosts the model performance. RTLCoder's lightweight property and low hardware barrier allow anyone to easily replicate and further improve based on our existing solution.

VI. Acknowledgement

This work is partially supported by the Hong Kong Research Grants Council (RGC) ECS Grant 26208723.

REFERENCES

[1] OpenAI, "GPT-4 Technical Report," *arXiv preprint arXiv:2303.08774*, 2023.

[2] 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," in *MLCAD Workshop*, 2023.

[3] 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.

[4] 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," *arXiv preprint arXiv:2309.10730*, 2023.

[5] 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.

[6] 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.

[7] 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.

[8] 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.

[9] 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.

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

[11] Y. Lu, S. Liu, Q. Zhang, and Z. Xie, "Rtllm: An open-source benchmark for design rtl generation with large language model," *arXiv preprint arXiv:2308.05345*, 2023.

[12] M. Liu, N. Pinckney, B. Khailany, and H. Ren, "Verilogeval: Evaluating large language models for verilog code generation," *arXiv preprint arXiv:2309.07544*, 2023.

[13] 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 *DATE*, 2023.

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

[15] M. Nair, R. Sadhukhan *et al.*, "Generating secure hardware using chatgpt resistant to cwes," *Cryptology ePrint Archive*, 2023.

[16] M. Rapp, H. Amrouch, Y. Lin, B. Yu, D. Z. Pan, M. Wolf, and J. Henkel, "Mlcad: A survey of research in machine learning for cad keynote paper," *IEEE TCAD*, 2021.

[17] K. Chang, K. Wang, N. Yang, Y. Wang, D. Jin, W. Zhu, Z. Chen, C. Li, H. Yan, Y. Zhou *et al.*, "Data is all you need: Finetuning llms for chip design via an automated design-data augmentation framework," *arXiv preprint arXiv:2403.11202*, 2024.

[18] Z. Pei, H.-L. Zhen, M. Yuan, Y. Huang, and B. Yu, "Betterv: Controlled verilog generation with discriminative guidance," *arXiv preprint arXiv:2402.03375*, 2024.

[19] E. Nijkamp, H. Hayashi, C. Xiong, S. Savarese, and Y. Zhou, "Codegen2: Lessons for training llms on programming and natural languages," *arXiv preprint arXiv:2305.02309*, 2023.

[20] R. Li, L. B. Allal, Y. Zi, N. Muennighoff, D. Kocetkov, C. Mou, M. Marone, C. Akiki, J. Li, J. Chim *et al.*, "Starcoder: may the source be with you!" *arXiv preprint arXiv:2305.06161*, 2023.

[21] A. Q. Jiang, A. Sablayrolles, A. Mensch, C. Bamford, D. S. Chaplot, D. d. l. Casas, F. Bressand, G. Lengyel, G. Lample, L. Saulnier *et al.*, "Mistral 7b," *arXiv preprint arXiv:2310.06825*, 2023.

[22] Y. Wei, Z. Wang, J. Liu, Y. Ding, and L. Zhang, "Magicoder: Source code is all you need," *arXiv preprint arXiv:2312.02120*, 2023.

[23] Y. Liu, P. Liu, D. Radev, and G. Neubig, "Brio: Bringing order to abstractive summarization," *arXiv preprint arXiv:2203.16804*, 2022.

[24] S. Bengio, O. Vinyals, N. Jaitly, and N. Shazeer, "Scheduled sampling for sequence prediction with recurrent neural networks," in *NeurIPs*, 2015.

[25] Synopsys, "VCS® functional verification solution," https://www.synopsys.com/verification/simulation/vcs.html, 2021.

[26] D. Guo, Q. Zhu, D. Yang, Z. Xie, K. Dong, W. Zhang, G. Chen, X. Bi, Y. Wu, Y. Li *et al.*, "Deepseek-coder: When the large language model meets programming–the rise of code intelligence," *arXiv preprint arXiv:2401.14196*, 2024.

[27] J. Rasley, S. Rajbhandari, O. Ruwase, and Y. He, "Deepspeed: System optimizations enable training deep learning models with over 100 billion parameters," in *KDD*, 2020.

[28] S. Takamaeda-Yamazaki, "Pyverilog: A python-based hardware design processing toolkit for verilog hdl," in *Applied Reconfigurable Computing*, 2015.