Improving LLM Performance in Generating Verilog by Fine Tuning with a Translated Code Dataset

Brendan Roberts

Electrical Engineering and Computer Sciences
University of California, Berkeley

Technical Report No. UCB/EECS-2025-104
http://www2.eecs.berkeley.edu/Pubs/TechRpts/2025/EECS-2025-104.html

May 16, 2025

Improving LLM Performance in Generating Verilog by Fine Tuning with a Translated Code Dataset

by Brendan Kyle Roberts

Research Project

Submitted to the Department of Electrical Engineering and Computer Sciences, University of California at Berkeley, in partial satisfaction of the requirements for the degree of **Master of Science, Plan II**.

Approval for the Report and Comprehensive Examination:

Committee:

Yakun Sophia Shao

Professor Sophia Shao
Research Advisor

5/15/2025

(Date)

* * * * * * *

Chris Fletcher

Professor Christopher Fletcher
Second Reader

5/15/2025

(Date)

Improving LLM Performance in Generating Verilog by Fine Tuning with a Translated Code Dataset

by
Brendan Kyle Roberts

A thesis submitted in partial satisfaction of the
requirements for the degree of
Master of Science
in
Electrical Engineering and Computer Sciences
in the
Graduate Division
of the
University of California, Berkeley

Advisor:

Professor Yakun Sophia Shao

Spring 2025

Abstract

Improving LLM Performance in Generating Verilog by Fine Tuning with a Translated Code Dataset

by Brendan Kyle Roberts

Master of Science in Electrical Engineering and Computer Sciences

University of California, Berkeley

Professor Yakun Sophia Shao, Advisor

Large language models (LLMs) are playing an increasingly large role in domains such as code generation, including hardware code generation, where Verilog is the key language. However, the amount of publicly available Verilog code pales in comparison to the amount of code available for software language like Python. In this work, we present hdl2v ("HDL-to-Verilog"), a dataset which seeks to increase the amount of available human-written Verilog data by translating or compiling three other hardware description languages—VHDL, Chisel, and PyMTL3—to Verilog. Furthermore, we demonstrate the value of hdl2v in enhancing LLM Verilog generation by improving performance of a 32 billion-parameter open-weight model by up to 23% (pass@10) in VerilogEvalV2, without utilizing any data augmentation or knowledge distillation from larger models. We also characterize and analyze our dataset to better understand which characteristics of HDL-to-Verilog datasets can be expanded upon in future work for even better performance.

Contents

List of Figures

List of Tables

Acknowledgements

First, I would like to thank my advisor, Professor Sophia Shao. I began research as a third-year in undergrad, and it is because of your continued support and guidance that I have made it to this milestone.

I also want to thank everyone I worked with while conducting research. To Charles, thank you for continuing to believe in me and supporting me along this journey. I will always be grateful for your mentorship, and it was an absolute pleasure working with you ever since I joined SLICE Lab. To Huijae, Alex, and Advay, thank you for your dedication to this project. Most importantly, though, I want to thank all of you for being great partners and friends. I'm wishing all of you the very best in your futures.

To my friends, thank you for standing by my side throughout graduate school. Your friendship has meant the world to me.

I want to thank my Mom, Dad, Gabriella, and everyone in my family for your unwavering support, and most of all for your unconditional love.

Finally, I want to acknowledge the role God has played in my life, He who continues to bless me as I finish graduate school and begin a new chapter in my life.

Chapter 1

Introduction

In recent years, Large Language Models (LLMs) have become nearly ubiquitous through the adoption of consumer facing LLMs such as OpenAI's ChatGPT and Google's Gemini. LLMs have demonstrated impressive performance in a wide range of tasks, ranging from general reasoning ability and instruction-following [1, 2] to code generation [3–6].

LLMs have potential to automate a wide range of tasks in hardware design, ranging from design to verification to optimization [7–9]. A number of studies have attempted to evaluate and improve LLMs' potential in generating Verilog code [10–12]. Compared to popular software programming languages such as Python or C, there is not as much publicly available Verilog. As of April 2025, there are 132,264 GitHub repositories with Python as the primary language, compared to just 848 for Verilog or SystemVerilog [13]. There is much less publicly available Verilog data, limiting our ability to effectively train LLMs in writing the language.

There are a variety of possibilities as to why there is less open-source Verilog data. Hardware development is much more niche compared to software, and companies that design hardware using Verilog keep their codebases private. This limits our ability to pretrain an LLM in writing Verilog, so alternative solutions such as fine tuning are necessary.

As a result, a number of prior works have attempted to fine-tune LLMs on novel Verilog datasets, and successfully improved Verilog generation performance. These works utilize techniques such as data augmentation [14–17] and synthetic Verilog generation [18].

However, Verilog is not the only hardware description language (HDL). VHDL is another popular HDL with its own ecosystem of supported hardware. While tehse languages are often used as the common interface between hardware code and tools such as RTL simulators or synthesis software,

design can be done in higher-level languages such as Chisel. Nevertheless, Verilog is the most important HDL to generate using LLMs as it is the most commonly supported and written.

In this work, we investigate how datasets of alternative HDLs can be used to improve an LLMs ability to write Verilog. Specifically, we present hdl2v, a dataset consisting of 46,549 pairs of VHDL, Chisel, and PyMTL3 translated/compiled to Verilog. We use this data to supervise fine-tune LLMs, and the results of these experiments are as follows:

- Fine-tuning with this data yields significant improvements in Verilog generation performance. We find that VerilogEvalV2 performance of a state-of-the-art open-weight LLM improves by up to 13% for pass@1 and 23% for pass@10 after being fine-tuned on a combination of our datasets.
- Language matters; fine-tuning with VHDL-Verilog pairs yields better results than C-Verilog pairs when the Verilog remains constant.
- Our fine-tuned models learn from the code in prompt-response pairs, not just natural language. However, utilizing meaningful module and variable names is important in helping LLMs learn from the data.

hdl2v is fully open-source and available for others to expand on this research. [1] [2] [3]

[1] VHDL dataset: `https://huggingface.co/datasets/rtl-llm/vhdl_github_deduplicated`
[2] Chisel dataset: `https://huggingface.co/datasets/rtl-llm/chisel-verilog-pairs`
[3] PyMTL dataset: `https://huggingface.co/datasets/rtl-llm/PyMTL_Verilog_pairs`

Chapter 2

Background

2.1 Verilog Code Generation

Prior work has seeked to improve LLMs' ability to generate correct Verilog code, using techniques such as fine tuning on textbooks and Verilog from Github [10]. Other prior works augment existing Verilog datasets [14–17] or generate novel synthetic Verilog [18]. Multi-agent systems utilize feedback from RTL simulation tools and modify test code in order to debug generated code [11, 12]. Benchmarks such as VerilogEval [19] and RTLLM [20] have been developed to standardize evaluation of LLM Verilog generation performance.

This work does not supercede such prior work. Instead, we seek to provide new data that can complement other approaches, such as data augmentation and agentic systems, to further improve LLM Verilog generation.

2.2 Verilog Translation

BetterV [15] introduces the idea of translating Verilog to C in order to improve correctness of generated Verilog. However, we demonstrate in Section 4.4 that when training with a Verilog dataset and the corresponding data translated to C, the benefit from training with C is minimal compared to training with only Verilog. Furthermore, the Verilog in the dataset is also likely to be present in LLMs' pre-training data, as it originates from open-source Github repositories.

To our knowledge, this work is the first to translate other HDLs to Verilog to generate novel Verilog data for fine-tuning.

Chapter 3

Dataset Construction

3.1 Datasets From Prior Work

As a baseline, we fine-tune with datasets from prior work that are based on existing Verilog from public sources.

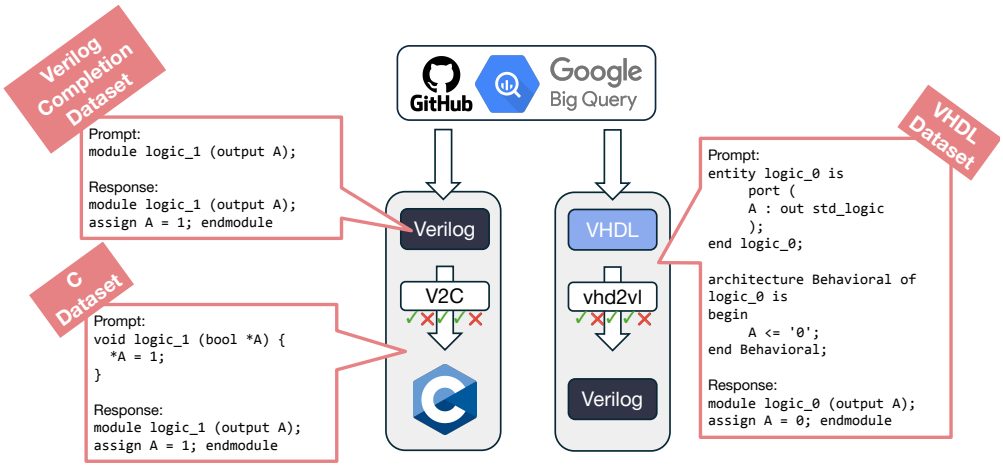

FIGURE 3.1: How the Verilog Completion, C, and VHDL datasets are collected. Note that Verilog is translated to C and VHDL is translated to Verilog, but during fine-tuning Verilog is always used as the response.

3.1.1 Verilog Completion

As in BetterV [15], this dataset consists of a filtered set of Verilog modules from public sources. In this case, the prompt for fine-tuning is the header of the Verilog module, and the response is the

rest of the module. Figure 3.1 shows an example of what an entry in this dataset might look like. This dataset contains 147,138 entries with total size 84.4 MB.

3.1.2 C

As in BetterV [15] and as depicted in Figure 3.1, we use v2c [21] to translate the above Verilog to C. In this case, the prompt for fine-tuning is actually the translated C code, and the original Verilog is the response. The intent is to improve the model's understanding of the Verilog code by attempting to correlate it with C code, which has greater presence in pretraining data. As not all Verilog modules in our dataset can be successfully translated to C, this results in 26,803 entries with size 36.5 MB.

3.2 hdl2v Datasets

Each hdl2v dataset is fully open-source and available on HuggingFace, as described in Section 1.

3.2.1 VHDL

As shown in Figure 3.1, we use Google BigQuery to collect VHDL files from Github. We filter by grabbing every file with a `.vhd` or `.vhdl` extension. This results in 53,698 VHDL entities. We attempt to translate each VHDL entity to Verilog using the open-source tool vhd2vl [22]. We are able to successfully translate 8974 of these entities to Verilog, with a total size of 51.5 MB.

3.2.2 Chisel

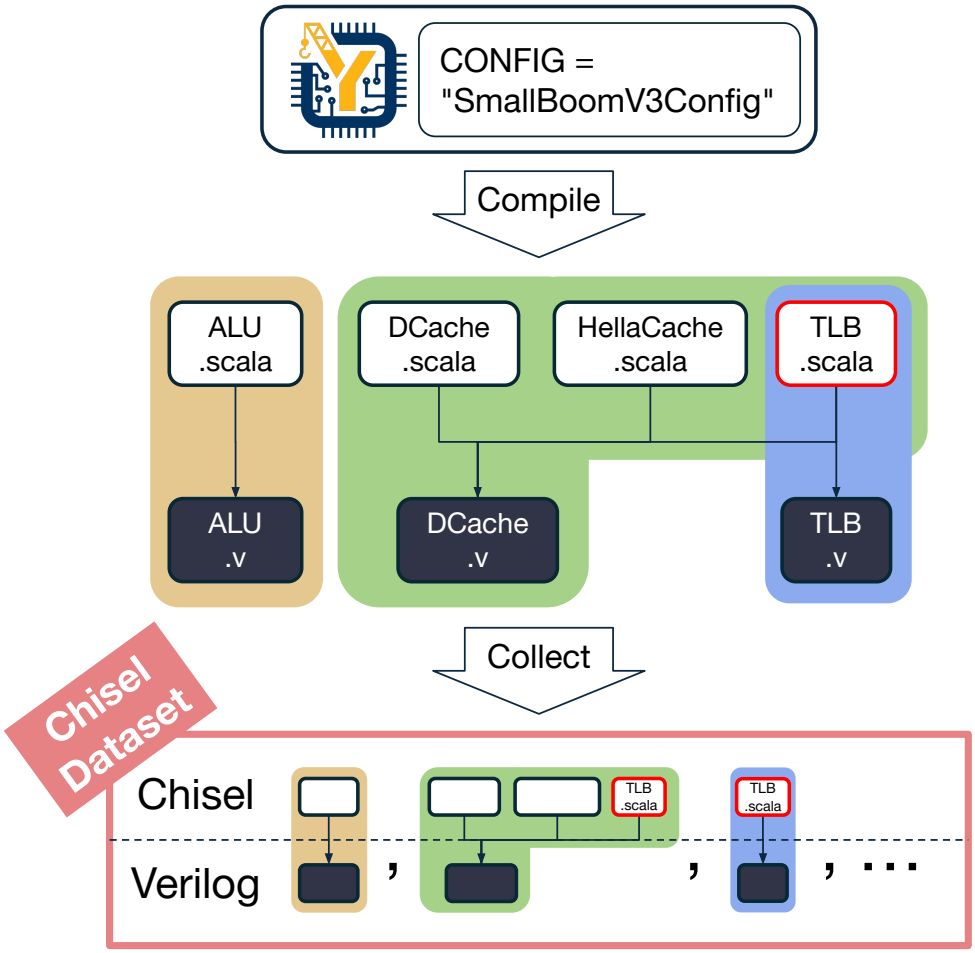

FIGURE 3.2: How the Chisel dataset is collected. One pair is collected for each generated Verilog module. Note that in cases where multiple Verilog modules are generated from the same Chisel file, that Chisel file will be included in multiple pairs in the dataset.

Chisel is a high-level HDL embedded in Scala that can be compiled into Verilog or SystemVerilog. Therefore, it intrinsically provides matching pairs between itself and Verilog. To gather this data, we used the Chipyard framework [23], which contains a variety of generators that can be combined to easily create top-level configurations for SoCs.

We compile a large number of Chipyard SoC configurations to Verilog, aiming to collect Verilog files pointing to every single Chisel source file in the default Chipyard repository. Our dataset includes 55% of the .scala files in Chipyard and its subrepositories; of those not covered, most do not contain synthesizable Chisel. The generated Verilog contains annotations that indicate which Chisel file and line each Verilog line is generated from, allowing us to collect a set of relevant Chisel files for each generated Verilog file. Each Verilog file contains one module. We show an example of how

Chisel-Verilog pairs are collected for one SoC configuration in Figure 3.2. Duplicates are removed, but for cases where the same Chisel file with different parameters generates differing Verilog output, all the data is kept.

This results in 18,939 Chisel/Verilog pairs (with a total size of 1.69 GB).

In our prompt/response pairs for LLM fine-tuning, the response consists of one generated Verilog file, which contain a single Verilog module. The prompt contains one or multiple Chisel source files (including one primary class and all of its dependencies), and a request to translate the code into Verilog. While the Chisel source code provided in the prompt does not directly compile to the response Verilog, we seek to improve the LLM's ability to correlate high-level HDL code in the Scala-embedded Chisel, which may contain more information about design semantics, to lower-level compiled Verilog.

3.2.3 PyMTL

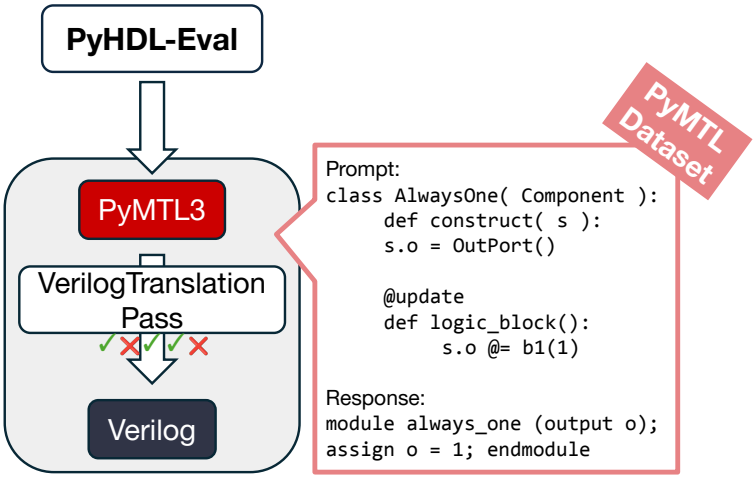

FIGURE 3.3: How the PyMTL dataset is collected.

PyHDL-Eval [24] evaluates the ability of LLMs to correctly generate the Python-embedded HDLs PyMTL3, PyRTL, MyHDL, Migen, and Amaranth. The authors provide as an artifact the PyHDL code generated by LLMs during these experiments.

As shown in Figure 3.3, we use LLM-generated PyMTL3 code from PyHDL-Eval to construct a dataset similar to the Chisel dataset above. Like Chisel, PyMTL3 is a high-level HDL which can be compiled to Verilog. We compiled each PyMTL3 example from PyHDL-Eval's artifact, numbering about 50,000, to Verilog using PyMTL3's VerilogTranslationPass. 18,636 examples (with a total size

of 28 MB) compiled to Verilog successfully, as many LLM-generated PyMTL3 examples contained syntax errors.

Chapter 4

Experiment Setup

4.1 LLM Fine-Tuning Setup

We use Qwen2.5-Coder-32B-Instruct [25] as our base model. To train the model, we use DeepSpeed-Chat's Supervised Fine-Tuning pipeline [26] and enable ZeRO Stage 3 [27] and LoRA [28] for efficient training. We maintain consistent hyperparameter settings across all experiments, including the use of FusedAdam optimizer, cosine learning decay, a learning rate of 1e-5, a single training epoch, and a batch size of 8. All experiments are conducted on a server with four NVIDIA L40S GPUs. A summary of the training hyperparameters is provided in the following table:

Hyperparameter	**Value**
ZeRO Stage	3
LoRA Dimension	32
Data Type	bfloat16
Batch Size	8
Learning Rate	1e-5
Number of Epochs	1

Table 4.1: Hyperparameters used for Fine-Tuning

4.2 Evaluation Setup

We use VerilogEvalV2 [29] to evaluate our model, with the following settings:

Parameter	Value
Samples	20
Temperature	0.85
top_p	0.95
ICL examples	0
ICL rules	no

TABLE 4.2: Parameters used for evaluation with VerilogEvalV2

4.3 Experiments

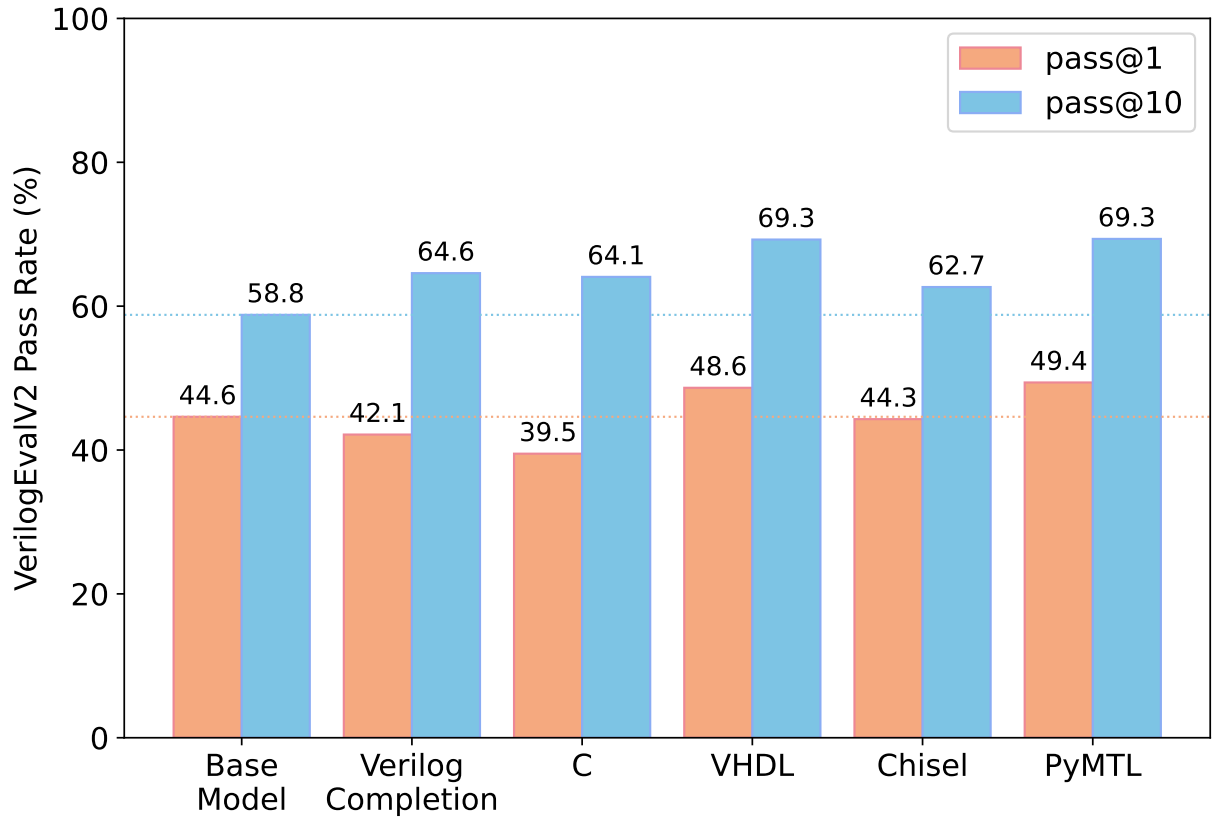

FIGURE 4.1: VerilogEvalV2 performance for Qwen2.5-Coder-32B-Instruct, after being fine-tuned with each individual dataset.

4.4 Fine-Tuning with Individual Datasets

As shown in Figure 4.2, our datasets have varying effectiveness in improving our models' performance in VerilogEvalV2. Of the five datasets tested, PyMTL and VHDL perform the best, both

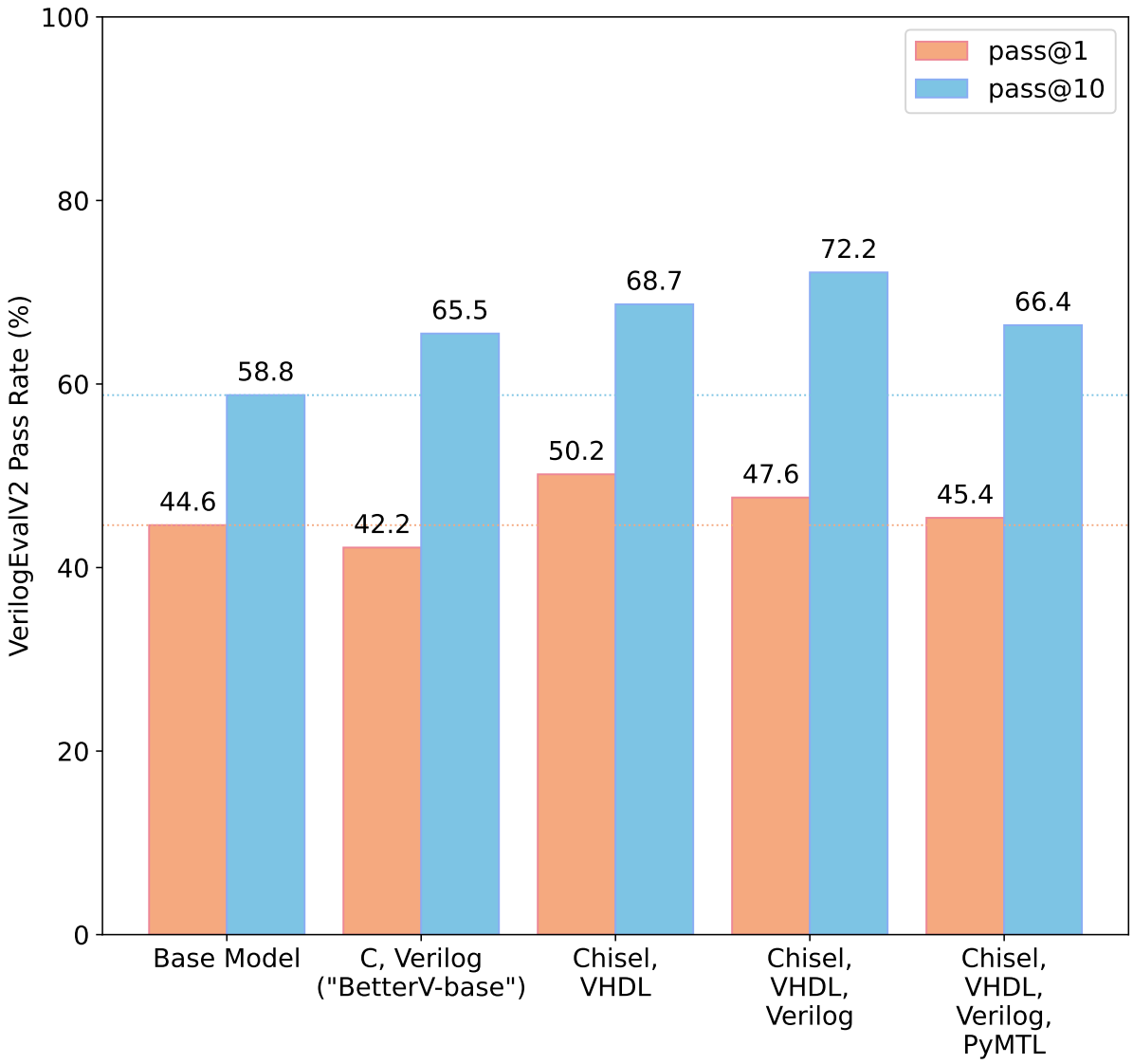

FIGURE 4.2: VerilogEvalV2 performance for Qwen2.5-Coder-32B-Instruct, after being fine-tuned with combined datasets.

providing about 18% increase in pass@10 over the base Qwen2.5-Coder-32B-Instruct model.

As discussed in Section 4.6, the VHDL dataset has the highest perplexity in the dataset and provides a diverse dataset that has not been seen (as Verilog) during pretraining. On the other hand, while the PyMTL dataset is relatively less diverse, the set of designs it targets is highly relevant to VerilogEval, as PyHDL-Eval also generated code for a benchmark set of designs similar to VerilogEval.

The Verilog, C, and Chisel datasets provide relatively smaller improvements. In fact, the C and

Verilog Completion datasets decrease pass@1, but increase pass@10, which indicates that fine-tuning on these datasets has increased the diversity of generated code.

4.5 Combining Datasets

We also explore the effects of fine-tuning with multiple datasets combined.

First, we fine-tune with our equivalent of BetterV's fine-tuning dataset. This includes the C dataset with both directions (C in the prompt and Verilog in the response, and vice versa), as well as our Verilog Completion dataset. Note that this data is not exactly the same as the dataset used in BetterV, and we do not include the discriminative guidance component.

Combining the C and Verilog datasets did not yield a significant improvement (within about one percentage point) compared to just Verilog. This is likely because these datasets originate from the same Verilog data, and as we show in Section 4.8, there are other languages that might perform better than C when translated to.

Next, we fine-tune data with other combinations of datasets. Specifically, we interleave datasets one entry at a time such that the distribution in the beginning of fine-tuning (when learning rate is highest) is equal for each dataset used. Combining datasets seems to yield limited but positive results. In particular combining Chisel and VHDL datasets yields our highest pass@1 of 50.2%, and combining Chisel, VHDL, and Verilog yields our highest pass@10 of 72.2%. However, combining Chisel, VHDL, Verilog, PyMTL reduces both pass@1 and pass@10.

In future work, we would like to explore other methods of composing these datasets in order to harness the strengths of all the datasets in a single trained model.

4.6 Analysis

Metric		**Verilog Completion**	**C**	**VHDL**	**Chisel**	**PyMTL**
Total Tokens		27,818,433	5,274,385	7,039,588	128,662,957	6,607,407
Vocabulary Size		23,247	15,075	22,279	4,441	1,731
Type-Token Ratio (TTR)		0.0008	0.0029	0.0032	0.0000	0.0003
N-gram Diversity	2-gram	0.0195	0.0393	0.0407	0.0003	0.0017
	3-gram	0.0767	0.1217	0.1032	0.0010	0.0046
Average Entry Length (no. tokens)		188.06	195.78	783.71	7394.64	353.55
Standard Deviation of Entry Length		279.89	438.75	1796.34	10804.84	284.06
Perplexity		1.81	2.15	2.34	1.55	1.84

Table 4.3: Statistics for individual datasets

Our datasets differ along several axes. In addition to language, they also vary in factors such as distribution of designs and human readability. In this section, we characterize our datasets and perform two ablation studies to better understand what makes a dataset helpful in improving Verilog generation.

4.7 Dataset Statistics

We characterize the **Verilog** code of each dataset to eliminate effects of language syntax. We use Qwen2Tokenizer to compute token counts and diversity. Table 4.3 includes statistics such as:

- **Type-Token Ratio (TTR)**, the ratio of unique tokens to total tokens.
- **N-gram diversity**, the ratio of unique token sequences of length N to the total number of such sequences. Higher values indicate greater token variety for both metrics.
- **Perplexity**, which measures how well a model makes predictions on a dataset, with lower values indicating better performance. The model's prediction accuracy can be estimated using the formula: $\frac{1}{\text{perplexity}} \times 100$. For example, a perplexity of 1.81 corresponds to a prediction accuracy of about 55.2%. In our case, we use Qwen2.5-Coder-7B as our model and randomly sample 1000 entries from each dataset to compute perplexity.

We do not notice a significant correlation between any of these metrics and usefulness for Verilog fine-tuning.

4.8 C vs VHDL

In Section 3.1.2, we created our C-Verilog fine-tuning dataset by translating Verilog to C. In order to isolate the effects of using different languages from other variables, we create a dataset of Verilog to VHDL translations using this same dataset. We translate Verilog to VHDL using Icarus Verilog [30]. Both C and VHDL datasets are machine-translated, and they sample the same distribution of designs.

We train models using the subset of 12,612 pairs that were able to be translated to both C and VHDL. Due to the high computational and runtime costs of fine-tuning, we use Qwen2.5-Coder-7B-Instruct (the 7B variant rather than 32B) as our base model. Figure 4.3 shows that the VHDL-translated dataset performs noticeably better than the C dataset. This indicates that the syntactic

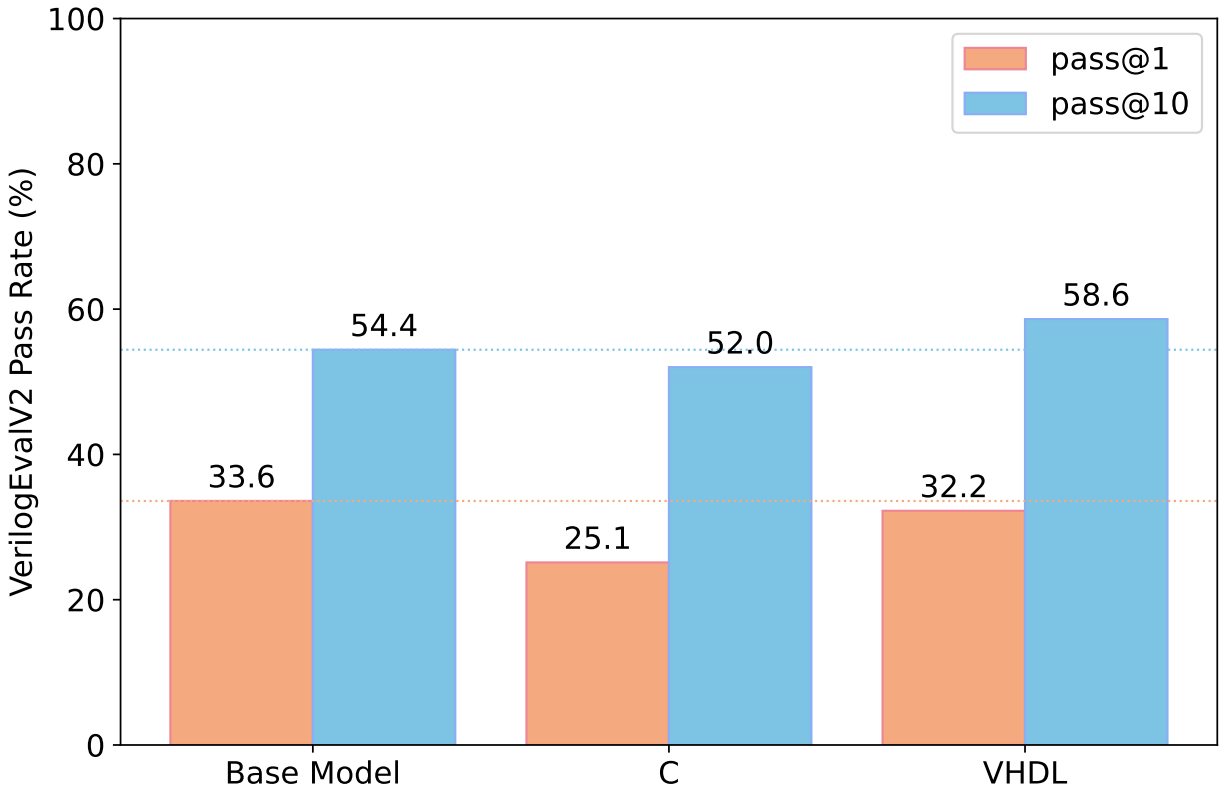

FIGURE 4.3: VerilogEvalV2 performance for Qwen2.5-Coder-7B-Instruct, after being fine-tuned with a subset of the Verilog dataset translated to C and VHDL, respectively.

closeness of VHDL to Verilog, and the fact that VHDL is an HDL (as opposed to C, which is a software language) plays some role in our VHDL dataset outperforming our C dataset.

4.9 Modifying the VHDL Dataset

In this section, we explore another axis of difference between datasets: human-readability. As a first step, we remove any comments from the VHDL dataset, then fine-tune Qwen2.5-Coder-7B-Instruct with this modified dataset. Next, in addition to removing comments, we obfuscate variable names across both VHDL and Verilog by replacing variable names with generic placeholders. VHDL/Verilog keywords are preserved and common variable names across a VHDL-Verilog pair remain identical, but obfuscated.

As shown in Figure 4.4, we find that removing comments has only a small effect, whereas obfuscating variable names significantly degrades model performance. This shows that our model learns mostly from code, and the effect of natural language descriptions in the code (in the form of comments) is minimal.

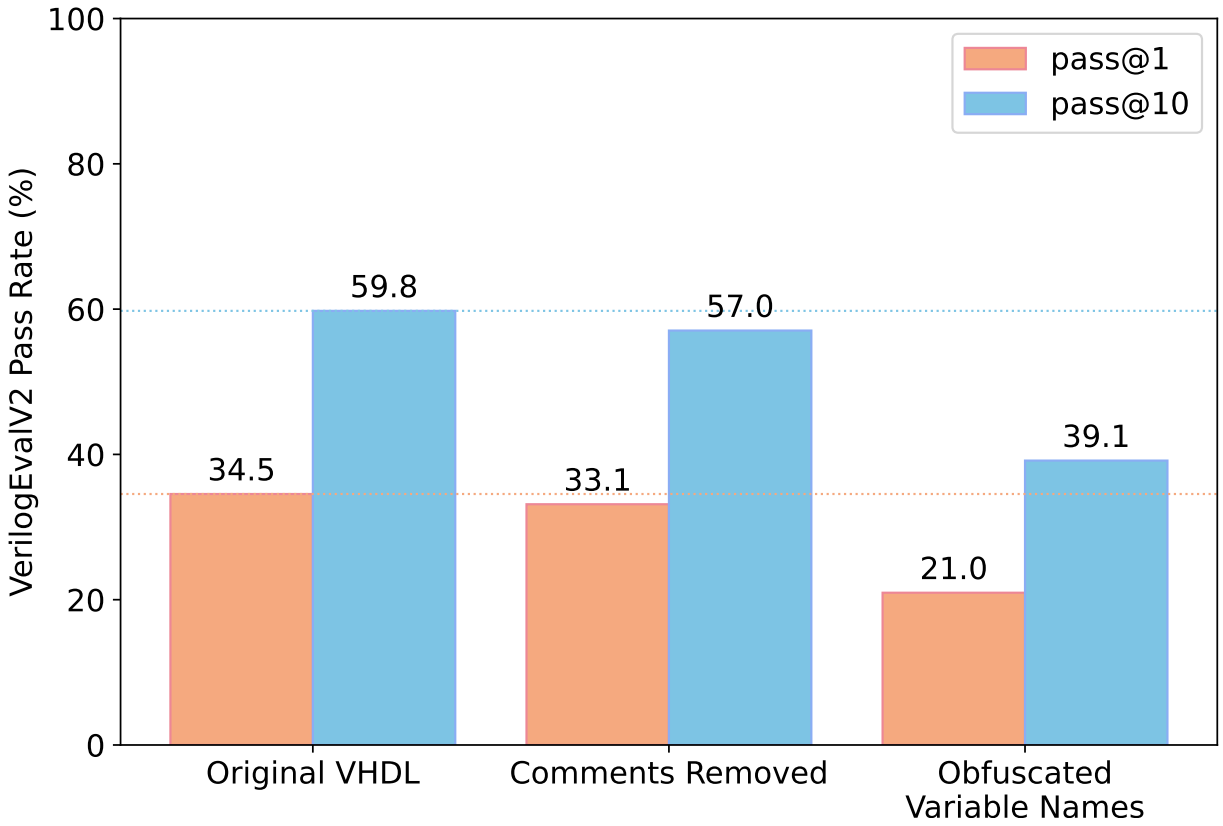

FIGURE 4.4: VerilogEvalV2 performance for Qwen2.5-Coder-7B-Instruct, after being fine-tuned with modified versions of our VHDL dataset.

4.10 Conclusion

In this work, we present hdl2v, which contains three new datasets for LLM Verilog generation fine-tuning. We utilize existing VHDL, Chisel, and PyMTL3 code to construct these datasets, and show that fine-tuning on these datasets yields up to 13% improvement in pass@1 and 23% improvement in pass@10 on VerilogEvalV2. Furthermore, we find that some languages are inherently better than others for this process; specifically we find that VHDL-Verilog pairs perform better than C-Verilog pairs for the same set of designs. We also find that the model does indeed learn from code rather than from the natural language comments in the code.

In future work, we would like to combine our dataset with other data augmentation methods and agentic flows to demonstrate that our methods can be composed with others and yield even greater improvement in Verilog generation performance.

Bibliography

[1] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners. In *Proceedings of the 34th International Conference on Neural Information Processing Systems*, NIPS '20, Red Hook, NY, USA, 2020. Curran Associates Inc. ISBN 9781713829546.

[2] OpenAI et al. Gpt-4 technical report, 2024.

[3] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.

[4] Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. Codegen: An open large language model for code with multi-turn program synthesis. *arXiv preprint arXiv:2203.13474*, 2022.

[5] Naman Jain, King Han, Alex Gu, Wen-Ding Li, Fanjia Yan, Tianjun Zhang, Sida Wang, Armando Solar-Lezama, Koushik Sen, and Ion Stoica. Livecodebench: Holistic and contamination free evaluation of large language models for code. *arXiv preprint arXiv:2403.07974*, 2024.

[6] Shishir G. Patil, Tianjun Zhang, Xin Wang, and Joseph E. Gonzalez. Gorilla: Large language model connected with massive apis. *arXiv preprint arXiv:2305.15334*, 2023.

[7] Jason Blocklove, Siddharth Garg, Ramesh Karri, and Hammond Pearce. Chip-chat: Challenges and opportunities in conversational hardware design. In *2023 ACM/IEEE 5th Workshop on Machine Learning for CAD (MLCAD)*, pages 1–6, 2023. doi: 10.1109/MLCAD58807.2023.10299874.

[8] Kaiyan Chang, Ying Wang, Haimeng Ren, Mengdi Wang, Shengwen Liang, Yinhe Han, Huawei Li, and Xiaowei Li. Chipgpt: How far are we from natural language hardware design, 2023. URL `https://arxiv.org/abs/2305.14019`.

[9] Charles Hong, Sahil Bhatia, Altan Haan, Shengjun Kris Dong, Dima Nikiforov, Alvin Cheung, and Yakun Sophia Shao. Llm-aided compilation for tensor accelerators. In *2024 IEEE LLM Aided Design Workshop (LAD)*, pages 1–14, 2024. doi: 10.1109/LAD62341.2024.10691748.

[10] Shailja Thakur, Baleegh Ahmad, Zhenxing Fan, Hammond Pearce, Benjamin Tan, Ramesh Karri, Brendan Dolan-Gavitt, and Siddharth Garg. Benchmarking large language models for automated verilog rtl code generation. In *2023 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 1–6, 2023. doi: 10.23919/DATE56975.2023.10137086.

[11] Yujie Zhao, Hejia Zhang, Hanxian Huang, Zhongming Yu, and Jishen Zhao. Mage: A multi-agent engine for automated rtl code generation, 2024. URL `https://arxiv.org/abs/2412.07822`.

[12] Chia-Tung Ho, Haoxing Ren, and Brucek Khailany. Verilogcoder: Autonomous verilog coding agents with graph-based planning and abstract syntax tree (ast)-based waveform tracing tool, 2025. URL `https://arxiv.org/abs/2408.08927`.

[13] URL `https://repositorystats.com/`.

[14] Yongan Zhang, Zhongzhi Yu, Yonggan Fu, Cheng Wan, and Yingyan Celine Lin. Mg-verilog: Multi-grained dataset towards enhanced llm-assisted verilog generation, 2024. URL `https://arxiv.org/abs/2407.01910`.

[15] Zehua Pei, Hui-Ling Zhen, Mingxuan Yuan, Yu Huang, and Bei Yu. Betterv: controlled verilog generation with discriminative guidance. In *Proceedings of the 41st International Conference on Machine Learning*, ICML'24. JMLR.org, 2024.

[16] Kaiyan Chang, Kun Wang, Nan Yang, Ying Wang, Dantong Jin, Wenlong Zhu, Zhirong Chen, Cangyuan Li, Hao Yan, Yunhao Zhou, Zhuoliang Zhao, Yuan Cheng, Yudong Pan, Yiqi Liu, Mengdi Wang, Shengwen Liang, Yinhe Han, Huawei Li, and Xiaowei Li. Data is all you need: Finetuning llms for chip design via an automated design-data augmentation framework. In *Proceedings of the 61st ACM/IEEE Design Automation Conference*, DAC '24, New York, NY, USA, 2024. Association for Computing Machinery. ISBN 9798400706011. doi: 10.1145/3649329.3657356. URL `https://doi.org/10.1145/3649329.3657356`.

[17] Yiyao Yang, Fu Teng, Pengju Liu, Mengnan Qi, Chenyang Lv, Ji Li, Xuhong Zhang, and Zhezhi He. Haven: Hallucination-mitigated llm for verilog code generation aligned with hdl engineers, 2025. URL `https://arxiv.org/abs/2501.04908`.

[18] Mingjie Liu, Yun-Da Tsai, Wenfei Zhou, and Haoxing Ren. Craftrtl: High-quality synthetic data generation for verilog code models with correct-by-construction non-textual representations and targeted code repair, 2025. URL https://arxiv.org/abs/2409.12993.

[19] Mingjie Liu, Nathaniel Pinckney, Brucek Khailany, and Haoxing Ren. Verilogeval: Evaluating large language models for verilog code generation, 2023. URL https://arxiv.org/abs/2309.07544.

[20] Yao Lu, Shang Liu, Qijun Zhang, and Zhiyao Xie. Rtllm: An open-source benchmark for design rtl generation with large language model. In *Proceedings of the 29th Asia and South Pacific Design Automation Conference*, ASPDAC '24, page 722–727. IEEE Press, 2024. ISBN 9798350393545. doi: 10.1109/ASP-DAC58780.2024.10473904. URL https://doi.org/10.1109/ASP-DAC58780.2024.10473904.

[21] R Mukherjee, M Tautschnig, and D Kroening. V2c – a verilog to c translator. volume 9636, pages 580–586. Springer, 2016.

[22] Larry Doolittle, Sep 2015. URL http://doolittle.icarus.com/~larry/vhd2vl/.

[23] Alon Amid, David Biancolin, Abraham Gonzalez, Daniel Grubb, Sagar Karandikar, Harrison Liew, Albert Magyar, Howard Mao, Albert Ou, Nathan Pemberton, Paul Rigge, Colin Schmidt, John Wright, Jerry Zhao, Yakun Sophia Shao, Krste Asanović, and Borivoje Nikolić. Chipyard: Integrated design, simulation, and implementation framework for custom socs. *IEEE Micro*, 40(4):10–21, 2020. ISSN 1937-4143. doi: 10.1109/MM.2020.2996616.

[24] Christopher Batten, Nathaniel Pinckney, Mingjie Liu, Haoxing Ren, and Brucek Khailany. Pyhdl-eval: An llm evaluation framework for hardware design using python-embedded dsls. In *Proceedings of the 2024 ACM/IEEE International Symposium on Machine Learning for CAD*, MLCAD '24, New York, NY, USA, 2024. Association for Computing Machinery. ISBN 9798400706998. doi: 10.1145/3670474.3685948. URL https://doi.org/10.1145/3670474.3685948.

[25] Binyuan Hui, Jian Yang, Zeyu Cui, Jiaxi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Keming Lu, Kai Dang, Yang Fan, Yichang Zhang, An Yang, Rui Men, Fei Huang, Bo Zheng, Yibo Miao, Shanghaoran Quan, Yunlong Feng, Xingzhang Ren, Xuancheng Ren, Jingren Zhou, and Junyang Lin. Qwen2.5-coder technical report, 2024. URL https://arxiv.org/abs/2409.12186.

[26] Zhewei Yao, Reza Yazdani Aminabadi, Olatunji Ruwase, Samyam Rajbhandari, Xiaoxia Wu, Ammar Ahmad Awan, Jeff Rasley, Minjia Zhang, Conglong Li, Connor Holmes, Zhongzhu Zhou, Michael Wyatt, Molly Smith, Lev Kurilenko, Heyang Qin, Masahiro Tanaka, Shuai

Che, Shuaiwen Leon Song, and Yuxiong He. Deepspeed-chat: Easy, fast and affordable rlhf training of chatgpt-like models at all scales, 2023. URL https://arxiv.org/abs/2308.01320.

[27] Samyam Rajbhandari, Jeff Rasley, Olatunji Ruwase, and Yuxiong He. Zero: Memory optimizations toward training trillion parameter models, 2020. URL https://arxiv.org/abs/1910.02054.

[28] Edward J. Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. Lora: Low-rank adaptation of large language models, 2021. URL https://arxiv.org/abs/2106.09685.

[29] Nathaniel Pinckney, Christopher Batten, Mingjie Liu, Haoxing Ren, and Brucek Khailany. Revisiting verilogeval: A year of improvements in large-language models for hardware code generation, 2025. URL https://arxiv.org/abs/2408.11053.

[30] Stephen Williams and Michael Baxter. Icarus verilog: open-source verilog more than a year later. *Linux J.*, 2002(99):3, July 2002. ISSN 1075-3583.