Thriving with HDLs in the age of LLMs

Mark Zakharov
mzakharo@ucsc.edu
UC Santa Cruz

Farzaneh Rabiei Kashanaki
frabieik@ucsc.edu
UC Santa Cruz

Jose Renau
renau@ucsc.edu
UC Santa Cruz

ABSTRACT

Large Language Models (LLMs) are revolutionizing the programming language landscape. This work explores the challenges and implications of integrating LLMs with Hardware Description Languages (HDLs), enabling the use of HDLs for which LLMs lack prior training.

Our work presents enables LLMs for multiple HDLs via HDLAgent. HDLAgent enhances LLMs by merging them with hardware compilers, simulating a human programmer's iterative process with compilers where the LLM-generated code is refined through compiler feedback. This integration demonstrates substantial improvements. For instance, DSLX's success rate jumps from 8% with GPT-4 Turbo alone to an 81% success rate.

1 INTRODUCTION

Recent advancements in Large Language Models (LLMs). Despite being in the early stages of development, these models assist beginners in generating code frameworks, enabling code completion, incorporating assertions, and documenting code, among other significant benefits.

Interacting with LLMs typically involves a prompt system like ChatGPT [8] or an autocompletion like Copilot [3]. In the simplest case, a zero-shot scenario, the LLM responds to a query without prior examples. When one code example is provided, it is termed a one-shot; with multiple examples, it becomes a few-shot scenario. Benchmarks such as HumanEval [2] often evaluate LLMs's ability to generate code under these conditions.

Additionally, LLMs can interface with external tools or the external world, forming an AI Agent. Agents involve a state machine that manages the LLM's workflow, guiding its interactions with information from external tools or other LLMs.

This work concentrates on the dynamics between Large Language Models (LLMs) and Hardware Description Languages (HDLs). HDLs often form a niche community, and existing models fail to recognize some HDLs. The non-Von Neumann architecture of these languages introduces additional challenges for LLMs in code generation. Addressing these issues is crucial for the hardware community to leverage LLMs' full potential. The authors propose that LLMs hold greater benefit for HDLs than for conventional languages like Python, primarily due to the limited availability of documentation and tutorials within the hardware design community. LLMs can lower the barrier to entry for new programmers by providing essential support in this specialized field. The focus of this work is the interaction of LLMs with Hardware Description Languages.

HDLAgent enhances LLMs by integrating them with hardware compilers, emulating a human programmer's iterative approach with a compiler. This process allows HDLAgent to refine LLM generated code using compiler feedback, leading to more accurate outputs. This work uses a few-shot mechanism for the main question and compiler feedback.

The creation of new programming languages needs to be improved due to the extensive data and time required for LLM training. With new languages needing more substantial training data and therefore experiencing integration delays, HDLAgent offers a solution by enabling the utilization of existing LLMs for newly designed languages.

Designing a new HDL or programming language faces additional hurdles in the LLM era. By leveraging LLMs' knowledge of programming, the adoption of new HDLs could become more straightforward. Our work demonstrates that with sufficient context and compiler feedback, LLMs can accurately respond to queries in unfamiliar HDLs, where traditional few-shot approaches consistently fail. For instance, using HDLAgent with GPT-4 Turbo shows an 81% success rate in HDLEval [5] when writing DSLX [4], a language where GPT-4 Turbo alone has only an 8% success rate. HDLAgent also improves LLMs with no training on an HDL.

HDLAgent enables the use of LLMs with new HDLs. Given LLMs' efficiency in teaching and introducing languages, our approach can significantly accelerate the growth of a user base for new languages. Our findings indicate that the accuracy of compiler feedback directly correlates with the performance of LLMs in this context.

2 RELATED WORK

To avoid a human-in-the-loop, a coding Agent can be applied to Verilog generation. The same coding Agent ideas with self-reflection, RAG, and grounding can be applied to Verilog. Concurrent most related works include AutoChip [9], RTLFixer [10], and HDLdebugger [12].

AutoChip [9] uses testbench feedback to ground the generated Verilog. It is similar to Self-Edit [13] and Self-Repair [7], but with a Verilog focus. Since the focus is simulation errors, there are no clear few-shot contexts like in HDLAgent, where a few-shot can be generated for each error/warning message.

RTLFixer [10] uses ReAct [11] for self-reflection, and RAG for grounding compiler errors. Similiarly, HDLAgent uses HDL descriptions and few-shot examples to guide code generation, and compiler fix samples to address compiler errors. These compiler fix samples resemble RTLFixer human-generated explanations for various error messages.

HDLDebugger [12] fine-tunes CodeLlama, but instead of fine-tuning like RTLCoder to generate better Verilog, it fine-tunes CodeLlama to fix code generation. HDLDebugger uses the compiler error messages to ground the generation, and applies it to the fine-tuned CodeLlama to fix the code. HDLDebugger is a different approach than when available (Publication August 2024) could be applied to HDLAgent in the steps to fix compiler errors. One issue is that it will require fine-tuning for each HDL. From the provisional paper, HDLDebugger does not seem to apply self-reflection.

3 EVALUATION

Figure 1 shows the overall results for the proposed HDLAgent when using GPT4-turbo. Four different HDLs are evaluated against four benchmark tests, with the exception of DSLX evaluated against HDLEval-pipe [5] due to its inability to arbitrarily pipeline. VH stands for VerilogEval-Human, VM stands for VerilogEval-Machine, HC stands for HDLEval-Comb, and HP stands for HDLEval-Pipe.

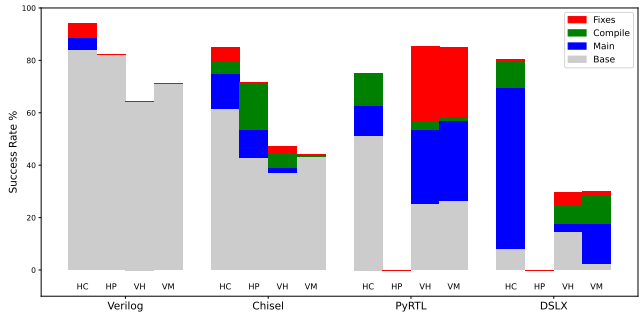

Figure 1: HDLAgent improves over the baseline across HDLs for GPT4-turbo.

Each bar has four components to showcase the impact of different aspects of HDLAgent. *Base* is the baseline or typical zero-shot LLM evaluation that does not use HDLAgent; *Main* adds the HDL Description as an initial context but it does not employ any compiler feedback. Main provides insights on the improvement by providing context and a few-shots related to the language used; *Compile* adds the compiler feedback and iterates up to 8 times to fix the code; *Fixes* performs the same iterations but for each iteration provides a suggestion, alongside a generic example, on how to fix that given compiler error.

To explain the HDLAgent overall results, we compare across LLMs for a given HDL at a time.

Verilog has the best overall performance for the *Base*, which is when no HDLAgent is active. This is expected behavior due to extensive training with Verilog syntax. An interesting observation is that HDLAgent can still slightly improve results, either by providing a main context or providing hints for code fixes (*Fixes*). There is a notable difference across benchmarks, with VerilogEval-Human emerging as the most challenging.

Chisel [1] is a significantly more interesting case. All the LLMs have significant training with Chisel, and even more so with Scala which is the base language upon which the Chisel DSL is built. The *Base* results without HDLAgent get close to 60% with HDLEval-Comb. This is notably lower than Verilog's results, which are around an 80% success rate. Chisel is a statically typed language, as such small issues are signaled by the compiler. These errors/warnings allow HDLAgent to iterate and improve the results and help match the performance for HDLEval-Comb.

For the other tests, Chisel does not match Verilog. This is expected for VerilogEval-Human because it includes Verilog specific questions not possible with Chisel. What is more problematic are the HDLEval-Pipe failures. Here Chisel starts with a lower base than Verilog, but it nearly matches Verilog once HDLAgent is enabled. HDLAgent is able to increase success rate by over 20%. This is significantly better but clearly pipelining is still an issue when compared with combinational logic.

PyRTL [6] is an HDL based on Python, but the LLMs analyzed have less training on PyRTL than Verilog and Chisel. This can be observed because Chisel outperforms PyRTL when no HDLAgent is enabled. Adding HDLAgent significantly improves the performance across HDLs. An intriguing finding from our results is the comparative performance of PyRTL and DSLX in the HDLEval-Comb evaluation. Specifically, DSLX achieves an 72% success rate, outperforming PyRTL's 60%.

This difference in improvement stems from PyRTL's nature as a generated language. PyRTL executes code to produce Verilog, causing the LLM to struggle to distinguish Python from PyRTL syntax. This confusion triggers more iterations and occasionally leads to semantically incorrect outputs. Conversely, DSLX, with its Rust-like syntax, presents closer semantic alignment. Although a significantly larger context could mitigate these issues, it introduces substantial costs and reduces speed. Large Language Models like GPT4 or CodeLlama, constrained to contexts under 16K tokens, cannot accommodate sufficient explanatory content in HDL descriptions. Chisel, which combines Scala and Chisel library syntax, also exhibits the same syntactic duality as PyRTL. However, this issue manifests little in Chisel due to extensive training of large LLMs with Chisel semantics. Consequently, HDLAgent does not require an extensive context to interpret and clarify Chisel syntax effectively.

DSLX [4] Since DSLX does not allow arbitrary pipelining, it can not be evaluated against HDLEval-Pipe and it exhibits very poor performance with VerilogEval. The reason is that several of the VerilogEval problems not only involve Verilog-specific questions but also necessitate using flip-flops or pipelines in nearly half of the cases. This implies that if DSLX was tested against VerilogEval problems it was capable of solving, the Success Rate % in Figures 4- 6 would be almost 2x greater. The result is that it is not fair to compare DSLX performance with VerilogEval. Hence the need for HDLEval as a language-neutral benchmark, as well as its breakdown between HDLEval-Comb and HDLEval-Pipe.

4 FUTURE WORK AND CONCLUSIONS

Large Language Models (LLMs) hold the potential to revolutionize many facets of computer science. This paper explores explicitly the application of LLMs in enhancing CHIP design realizations. Typically, CHIP design employs popular hardware description languages (HDLs) such as Verilog and Chisel, but there also exist several other less popular HDLs like PyRTL and DSLX. One of our contributions is developing an AI Agent named HDLAgent, which significantly improves LLMs' state-of-the-art HDL code generation, particularly for those with limited HDL proficiency. Supporting multiple HDLs without needing LLM tuning is crucial for pioneering new HDLs that exploit the capabilities of LLMs.

Overall, HDLAgent is able to improve the performance of unfamiliar HDLs across several LLMs. When comparing the results of combinational logic tests (HDLEval-Comb), the four HDLs evaluated perform relatively similarly. The evaluation section also provides multiple insights, such as developing a chat-like iterative context does not provide much benefit over a delta approach in terms of quality of results.

REFERENCES

[1] Jonathan Bachrach, Huy Vo, Brian Richards, Yunsup Lee, Andrew Waterman, Rimas Avižienis, John Wawrzynek, and Krste Asanović. Chisel: constructing hardware in a scala embedded language. In *DAC Design Automation Conference 2012*, pages 1212–1221. IEEE, 2012.

[2] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert,

Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. Evaluating large language models trained on code. 2021.

[3] GitHub Inc. Github copilot. https://copilot.github.com, 2023.

[4] Google. XLS Website. https://github.com/google/xls/, 2022.

[5] UCSC MASC Group. HDEval: Hardware Description Evaluation. https://github.com/masc-ucsc/hdeval, 2024. Online; accessed on April 2024.

[6] Diba Mirza, Deeksha Dangwal, and Timothy Sherwood. Pyrtl in early undergraduate research. In *Proceedings of the Workshop on Computer Architecture Education*, WCAE'19, New York, NY, USA, 2019. Association for Computing Machinery.

[7] Theo X. Olausson, Jeevana Priya Inala, Chenglong Wang, Jianfeng Gao, and Armando Solar-Lezama. Is self-repair a silver bullet for code generation?, 2023.

[8] OpenAI LLC. Openai. https://www.openai.com, 2023.

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

[10] Yun-Da Tsai, Mingjie Liu, and Haoxing Ren. Rtlfixer: Automatically fixing rtl syntax errors with large language models, 2024.

[11] Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. React: Synergizing reasoning and acting in language models, 2023.

[12] Xufeng Yao, Haoyang Li, Tsz Ho Chan, Wenyi Xiao, Mingxuan Yuan, Yu Huang, Lei Chen, and Bei Yu. Hdldebugger: Streamlining hdl debugging with large language models, 2024.

[13] Kechi Zhang, Zhuo Li, Jia Li, Ge Li, and Zhi Jin. Self-edit: Fault-aware code editor for code generation. In Anna Rogers, Jordan Boyd-Graber, and Naoaki Okazaki, editors, *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 769–787, Toronto, Canada, Jul. 2023. Association for Computational Linguistics.