

UC Santa Cruz

UC Santa Cruz Electronic Theses and Dissertations

Title

HDLEval a Benchmark for LLM-Driven RTL Design Using HDLAgent

Permalink

<https://escholarship.org/uc/item/53c255nr>

Author

Rabiei Kashanaki, Farzaneh

Publication Date

2024

Copyright Information

This work is made available under the terms of a Creative Commons Attribution License, available at <https://creativecommons.org/licenses/by/4.0/>

Peer reviewed|Thesis/dissertation

UNIVERSITY OF CALIFORNIA
SANTA CRUZ

**HDLEVEL A BENCHMARK FOR LLM-DRIVEN RTL DESIGN
USING HDLAGENT**

A dissertation submitted in partial satisfaction of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE AND ENGINEERING

by

Farzaneh Rabiei

June 2024

The Dissertation of
Farzaneh Rabiei is approved:

Professor Jose Renau, Chair

Professor Dustin Richmond

Professor Yi Zhang

Peter Biehl
Vice Provost and Dean of Graduate Studies

Copyright © by

Farzaneh Rabiei

2024

Table of Contents

List of Figures	v
List of Tables	vi
Abstract	vii
Acknowledgments	viii
1 Introduction	1
1.1 Introduction	1
2 Related Works	5
2.1 Related Work	5
3 HDLEval	9
3.1 HDLEval	9
3.1.1 Multi-HDL tests	9
3.1.2 Test Source	11
3.1.3 Test Classification	11
3.1.4 Test Evaluation	13
4 Setup	14
4.1 Setup	14
5 Evaluation	16
5.1 Evaluation	16
5.1.1 Overall Results	16
5.1.2 HDLAgent Context Insights	21
5.1.3 Pass Sensitivity	22
5.1.4 Time and QoR	24
5.1.5 Self-Reflection with Multi-Agent Debate	27
5.1.6 Insights for HDLs at the age of LLMs	29

6 Future Work and Conclusions	34
6.1 Future Work and Conclusions	34
Bibliography	36

List of Figures

5.1	HDLAgent improves Chisel across all LLMs.	17
5.2	HDLAgent improves PyRTL across all LLMs.	18
5.3	HDLAgent improves DSLX HDLEval-Comb across all LLMs.	19
5.4	Verilog succeeds across benchmarks and LLMs	20
5.5	HDL description and few-shot help LLMs to improve results.	22
5.6	GPro-1.0 converges in a few iterations.	24
5.7	LLM and HDL affect total HDLAgent execution time.	25
5.8	QoR is consistent across LLMs but different across HDLs.	26
5.9	Introducing a testbench into the generative loop worsens results	27

List of Tables

4.1	Language Tools and Versions	14
4.2	LLMs used in the evaluation	15
5.1	Pass@k results for HDLEval-Comb for different LLMs with just a Base query or with HDLAgent.	23

Abstract

HDLEval a benchmark for LLM-Driven RTL Design using HDLAgent

by

Farzaneh Rabiei

Large Language Models (LLMs) are transforming the programming language domain by facilitating learning for beginners and enhancing code generation and documentation. This paper delves into the challenges and potential of integrating LLMs with Hardware Description Languages (HDLs), specifically those HDLs that LLMs have not been previously trained on.

Our research offers two main contributions: the development of HDLAgent, which enables LLMs to work with multiple HDLs, and the creation of HDLEval, a language-neutral benchmark for HDLs. HDLAgent is an AI agent optimized for LLMs with limited prior knowledge of various HDLs, significantly enhancing the performance of off-the-shelf LLMs. For example, PyRTL’s success rate increases from zero to 35% with Mixtral 8x7B, and Chisel’s success rate improves from zero to 59% with GPT-3.5-turbo-0125. Meanwhile, HDLEval provides a flexible benchmarking system for multiple HDLs, adaptable to both current and future languages. Together, HDLAgent and HDLEval offer a robust framework for promoting the adoption and expanding the user base of HDLs in the age of LLMs.

Acknowledgments

I would like to express my deepest gratitude to Professor Jose Renau for his unwavering support and guidance over the past two years. His mentorship and the opportunity he provided for me to embark on this unforgettable journey have been invaluable.

This endeavor would not have been possible without the tireless efforts of Mark Zakharov, with whom I have had the privilege to collaborate for nearly a year. His development of the HDLagent was crucial to the success of this project.

I am also deeply thankful to Professors Dustin Richmond and Yi Zhang for their continuous support, mentorship, and invaluable industry advice.

Finally, I extend my heartfelt thanks to my husband. Without his help, I could not have proceeded. His strength, encouragement, and inspiration have fueled my passion for the wonderful field of computing.

Chapter 1

Introduction

1.1 Introduction

Recent advancements in Large Language Models (LLMs), such as OpenAI's GPT, Google's Gemini, and Mistral AI's Mixtral, are revolutionizing the field of programming languages. LLMs are making it easier for newcomers to learn and use programming languages by providing intelligent assistance, generating code snippets, and offering context-aware suggestions. LLMs have the potential to significantly reduce the entry barriers to programming.

The contribution of this work is to enable LLMs to operate with various Hardware Description Languages (HDLs). HDLs often form niche communities where existing LLMs may underperform. Although Verilog remains the most popular HDL, it is showing its age¹ and languages like Chisel3 [3], PyRTL [6], and DSLX [11] are being

¹The original Verilog was designed in 1983, and modern versions like System-Verilog are semantically compatible with it.

designed as alternatives. Unless off-the-shelf LLMs can work with new HDLs, there is a significant disincentive to create new HDLs.

Some of the most popular and high-performing LLMs are closed-source. This means that to make HDLs accessible to a wider community, the solution must work with LLMs that cannot update their training. Given the significant impact of LLMs, successful HDLs should support off-the-shelf models without requiring years of waiting for the next LLM training cycle to potentially incorporate the new HDL. If LLMs can handle new HDLs, they can facilitate easier adoption. Conversely, if LLMs cannot handle new HDLs, they become an additional obstacle that impedes the adoption of new HDLs.

The key contribution of this work is to enable the effective utilization of both open-source and proprietary LLMs across a diverse range of HDLs. This specifically addresses the challenge of generating accurate and functional code in HDLs that have proven difficult for existing LLMs.

To achieve the contributions outlined in the paper, we propose an AI Agent (**HDLAgent**) that incorporates state-of-the-art AI coding agent techniques, specifically adapted to support HDLs with limited LLM support. To evaluate the effectiveness of HDLAgent, we also introduce a new set of benchmarks (**HDLEval**). This is necessary because the current testing infrastructure for RTL design is tailored specifically to the Verilog language and is not suitable for assessing performance across multiple HDLs.

AI Agents [41] utilize multiple LLM queries and external tools, employing a state machine to manage the LLM’s workflow and guide its interactions. Many agents

leverage three main techniques: self-reflection, Retrieval Augmented Generation (RAG), and grounding. Self-reflection techniques like Chain-of-Thought (CoT) [30] use LLMs to improve their own responses. RAG provides context or examples for queries. Grounding uses external tools to reduce hallucinations or mistakes.

HDLAgent incorporates these state-of-the-art AI coding agent techniques but adapts them to the challenges of supporting HDLs. The self-reflection adaptations are tailored to manage module inputs/outputs specific to each HDL. RAG includes an HDL summary, provides few-shot examples, and compile-error fix examples for LLMs that may have limited knowledge of the given HDL. Grounding leverages compiler feedback to improve accuracy and reliability.

HDLAgent uses context (RAG) and compiler feedback (grounding) to carefully formulate queries for unfamiliar HDLs, where traditional few-shot approaches consistently fail. For instance, using HDLAgent with Mix-8x7B yields a 44% success rate when writing Chisel and just 3% tests pass without HDLAgent. Other LLM like GPT-3.5o, go from 3% success rate for DSLX to 48% with HDLAgent. HDLAgent also benefits LLMs with Verilog. For Mix-8x22B, the success rate goes from 13% to 53%.

LLMs are evaluated using benchmarks like HumanEval [4] to quantify the LLMs' coding capacity. Since HumanEval tests Python, a more recently proposed HumanEval-X [39] extends the tests to cover multiple languages. HumanEval-X creates a test for each supported language to verify the correctness of a response, but it focuses only on popular languages, not HDLs. VerilogEval [17] follows the HumanEval principle, but it employs Verilog tests instead of Python.

HDLEval supports multiple HDLs. Unlike HumanEval-X, which has language-specific problems, HDLEval’s problems are written purely in English, using the same inputs and tests across all HDLs. This allows one to leverage the same tests for future HDLs. Since not all HDLs support arbitrary pipelining, HDLEval also categorizes tests into Combination and Pipelining categories. Consequently, HDLEval facilitates the comparison of performance across different HDLs and LLMs simultaneously.

Additionally, HDLEval utilizes formal verification, instead of unit tests, to validate code generation. This allows for testing translation between languages, a difficult task in popular languages but readily available in HDLs.

HDEval includes tests similar in scope to VerilogEval but adapted to various HDLs. Additionally, it features several tests from recent Efabless [8, 9] competitions, where design teams could only use LLMs to implement tapeouts. We adapt the contestants’ queries to create larger and more challenging problems.

In summary, the paper contributions are HDLAgent and the HDLEval testing benchmark. HDLAgent enhances the performance of multiple HDLs across multiple LLMs. It shows that HDL design can leverage LLMs even before training examples or data is available. HDLEval allows to benchmark across HDLs and LLMs and provides insights and challenges that must be addressed. The evaluation also provides several insights that future HDLs designers to avoid LLM support issues.

Chapter 2

Related Works

2.1 Related Work

There are two main approaches to improve LLM output: Fine-tuning and Retrieval Augmented Generation (RAG). These techniques can be iteratively combined to develop Agents that produce even better results.

Fine-tuning is the process of adjusting the parameters of a LLM on a specific dataset or task to improve its performance. As such, fine-tuning can be applied to optimize a given LLM for a new language. RTLCoder [18] fine-tunes a 7B mistral model with GPT generated synthetic Verilog data. HDLAgent uses off-the-shelf LLMs without fine-tuning.

URIAL [14] bypasses the need for fine-tuning by enriching prompts with illustrative examples. These prompts resemble the few-shot format used by HDLAgent, incorporating both format and examples. While URIAL has shown effectiveness in

circumventing the need for instruction alignment, HDLAgent further illustrates the possibility of learning previously unknown languages.

Agents [41] iterate through LLMs with three main techniques to improve performance: Self-reflection, few-shot with RAG, and grounding.

Self-reflection techniques use a sequence of interactions with the LLM instead of a simple question/answer. CoT [30] is an example of self-reflection. Lumos [37] uses CoT to enable simpler LLMs to outperform more advanced LLMs. These studies highlight some significant progress in this rapidly evolving field. Recent works [33] propose an optimization method to find the best prompt.

Few-shot in-context learning [2, 15] with RAG [13] employs instructions and several examples (few-shot) related to the question or prompt to enhance efficiency. Various techniques exist for constructing such prompts with an extended context. This technique of querying an embedding database to augment the context is known as Retrieval Augmented Generation (RAG).

Grounding involves verifying or checking the LLM’s response using an external tool. While this isn’t always feasible, in code generation, a compiler or testbench can validate and find issues with the LLM-generated response. This triggers further iterations with the LLM.

Agents with self-reflection, RAG, and grounding have been applied to improve code generation. If we ignore the HDL target, and focus on generic programming languages like Python or C++, several works [23, 40] show that many errors can be fixed by grounding the generated code against a compiler or test. Recent works [7, 19–

21, 28, 32] leverage this fact and propose Agents that ground the code generation with a compiler or tests. Other works [10, 22, 38] propose Agents to iterate over test bench results to fix the generated code.

Besides CoT, some notable self-reflection for code generation include: Self-planning [12] proposes a planning stage or self-reflection before code generation; Self-Debug [5] proposes how to improve code generation by generating explanations in the intermediate steps; ChatCoder [29] uses self-reflection to paraphrase and elaborate on the initial question.

Early work [24, 25, 34] with LLMs and Verilog avoid the Agents because LLMs like GPT-4 are already reasonably trained for Verilog. Several AI-based chip design competitions [8, 9] required designs implemented in Verilog. Looking at the top performers, they tend to use GPT-4 and focus on combinational modules where the top level module IO is fully specified. In all the cases, the human-in-the-loop guides the LLM to fix problems with the generated code and iterate over the testbench results.

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 works include AutoChip [26], RTLFixer [27], and HDLdebugger [36].

AutoChip [26] uses testbench feedback to ground the generated Verilog. It is similar to Self-Edit [38] and Self-Repair [22], 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 [27] uses ReAct [35] for self-reflection, and RAG for grounding compiler errors. Similarly, 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 [36] fine-tunes CodeLlama, but instead of fine-tuning like RTL-Coder 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.

Chapter 3

HDLEval

3.1 HDLEval

HDLEval aims to develop a comprehensive suite of tests applicable to various HDLs, offering insights into the challenges associated with both LLMs and HDLs. The authors intend to continuously release new HDLEval versions and make it available as an open-source resource.

3.1.1 Multi-HDL tests

The primary innovation of HDLEval lies in its establishment of an HDL agnostic testing framework. At first glance, VerilogEval suffices as a comprehensive test since all HDLs ultimately translate into Verilog. However, the issue is that VerilogEval encompasses a set of tests tailored explicitly for Verilog.

For instance, a basic test in VerilogEval instructs the implementation of a D

latch using an "always" block. This construct poses a challenge as certain HDLs do not support latch implementation, and more critically, many non-Verilog HDLs lack the concept of "always" statements.

HDLEval's methodology involves providing a purely English description of the problem along with inputs and outputs. A significant challenge is the matching of inputs and outputs. IO matching is an issue because different languages utilize various styles. Some use wires, others adopt a struct-like approach, while others employ interfaces. All HDLs generate Verilog, so we ensure the HDL-generated Verilog and the testing Verilog IOs match with some heuristics. For example, when there is a single output, we create wrappers that ensure that the output names are identical. Additionally, we do not use IO order; matching is only by name.

Another critical aspect is the range of functionality supported by different languages. Verilog could be seen as the 'assembly language' of HDLs, capable of describing all digital designs. However, not all languages offer the same feature set. For instance, DSLX lacks support for arbitrary pipelining constructs. Instead, it employs an actor-like syntax for pipelines communicating via elastic (valid/stop) signaling. Therefore, pipelining tests would invariably fail in DSLX, a limitation not of the LLM or HDL but of the applicability of specific tests.

Our solution involves categorizing tests based on their functionality, ensuring that only those compatible with a given HDL are included. HDLEval tests have attributes such as "combinational" and "pipelining" to facilitate this process.

3.1.2 Test Source

We derive HDLEval tests from three principal sources: HDLBits, custom tests, and Efabless competitions.

HDLBits serves as a pedagogical platform for introducing students to Verilog. A significant portion of our VerilogEval tests originate from this website. We selectively incorporate these tests, excluding those that are exclusively pertinent to Verilog syntax. In designing tests, we ensure compatibility across multiple hardware description languages (HDLs), including Verilog, Chisel, pyRTL, and DSLX, by maintaining uniform problem descriptions and interface definitions.

Custom tests, developed by our research group’s students, address topics not encompassed by HDLBits and incorporate more complex design elements. For instance, one custom test requests is a 8-bit floating point addition unit.

Efabless competitions constitute another source of HDLEval tests. We adapt published prompts into new tests, often refining the problem through iterative question adjustments aimed at resolving specific issues. Given HDLEval’s HDL-agnostic nature, we remove intermediate Verilog solutions and modify the prompts to reach an equivalent code. Though subjective, this approach facilitates the creation of tests that push the capabilities of current language models.

3.1.3 Test Classification

Upon analyzing various LLMs and tests, we noted that LLMs well-versed in specific HDLs generally perform effectively with combinational logic but encounter dif-

difficulties with pipelining tests. This distinction arises because combinational logic aligns more closely with traditional Von Neumann or non-hardware language structures. When a hardware designer articulates a problem using a non-hardware language like C, excluding pointers and runtime loops, the resulting syntax and semantics can often directly translate into a valid combinational logic module. However, this direct translation is not feasible for pipelining.

This trend is also evident in the recent Efabless competitions [8,9], where the majority of participating teams queried for pre-designed combinational blocks, deliberately avoiding complex pipelining within these blocks apart from basic output flopping. Based on our experience, introducing arbitrary pipelining reduces the likelihood of success.

These observations and the fact that some languages lack support for arbitrary pipelining constructs have led us to bifurcate HDLEval into two distinct categories: Combinational (HDLEval-Comb) and Pipelining (HDLEval-Pipe). The Combinational category consists of a set of tests without flip-flops. Current LLMs with extensive programming knowledge can successfully generate these. On the other hand, the Pipelining category includes arbitrary pipelining and presents a set of more challenging problems.

This division will streamline the testing process and highlight specific challenges in HDLs/LLM pipelining performance that future research needs to address.

3.1.4 Test Evaluation

A novelty from HDLEval is the use of logic-equivalence-check (LEC) instead of a testbench. This is not a problem in HDLs because all the existing HDLs generate Verilog, and existing open-source tools like Yosys [31] can perform LEC between modules. HumanEval also has this problem. The recent EvalPlus [16] shows that bugs injected into HumanEval are not always captured by the HumanEval tests. A LEC step is a formal equivalence that a small set of tests can not prove.

A challenge in using LEC is its requirement for convergence to affirm the equivalence of two modules. This issue did not happen in any VerilogEval and HDLEval combinational designs. However, in our examination of VerilogEval tests, we encountered an instance with pipelining where the LEC failed to complete its analysis. This occurred in a test involving a state machine designed to calculate population count.

For some pipeline tests, the LEC step can not prove equivalence or failure, but we find that this is not a problem. The reality is that LEC tries many combinations of values and none fail. From a LEC point of view, there are too many potential values to cover and hence it can not prove correctness. This is in fact a more powerful test than most HumanEval assertion tests. This rare scenario occurs only once in 156 tests. In HDLEval, we consider passing the test unless LEC reports a mismatch.

Chapter 4

Setup

4.1 Setup

Table 4.1 lists all the languages used in the evaluation and the compiler versions used by this paper. When a date is provided it corresponds to the top-of-tree version at that given month. For Quality of Results (QoR), we use Yosys synthesis results.

Table 4.1: Language Tools and Versions

Language	Tool	Version
Verilog	Yosys	0.35
Chisel	FIRRTL	3.5.0-RC2
pyRTL	pyRTL compiler	0.10.2
DSLX	XLS	3/2024

Table 4.2 shows the LLMs used. Many LLMs, including GPT-3.5o, are not deterministic. They have produced differing outcomes for the same example under identical prompt conditions. OpenAI recently proposed a new API to address this issue, providing a seed, but this solution still needs to be fully implemented across all

Table 4.2: LLMs used in the evaluation

LLM	Version	Updated	Context
GPT-4	gpt-4-1106-preview	4/2023	128000
GPT-3.5n	gpt-3.5-turbo-0125	9/2021	16385
GPT-3.5o	gpt-3.5-turbo-1106	9/2021	16385
GPro-1.0	gemini-1.0-pro-001	2/2024	32720
Mix-8x7B	Mixtral-8x7B-instruct-v0.1	12/2023	32768
Mix-8x22B	Mixtral-8x22B-v0.1	3/2024	32768

LLMs. For fair evaluation, we avoid the deterministic settings and perform 1, 5, or 10 runs depending on the pass@k parameter.

Chapter 5

Evaluation

5.1 Evaluation

5.1.1 Overall Results

To understand the benefits of HDLAgent across LLMs, we plot each of the HDLs (Chisel, PyRTL, DSLX, and Verilog) against four benchmark tests: VH stands for VerilogEval-Human, VM stands for VerilogEval-Machine, HC stands for HDLEval-Comb, and HP stands for HDLEval-Pipe.

VerilogEval tests consist of several Verilog-specific questions. However, their evaluation does not fully demonstrate the potential of the LLM/HDLAgent as effectively as HDLEval (HC, HP) does. This is a common issue across HDLs with the exception of Verilog. The plots include VH and VM for reference, but the focus of the evaluation is HDLEval-Comb.

Each bar has five 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 yet still keeps the HDLAgent prompts to format IOs and overall directions for code generation; *Description* adds the HDL Description (Section ??); *Few-shot* adds the few-shot related to the language used; *Compile* adds the compiler feedback and iterates up to 8 times to fix the code; and *Fixes* performs the same iterations but for each iteration provides a suggestion, alongside a generic example, on how to fix that given compiler error.

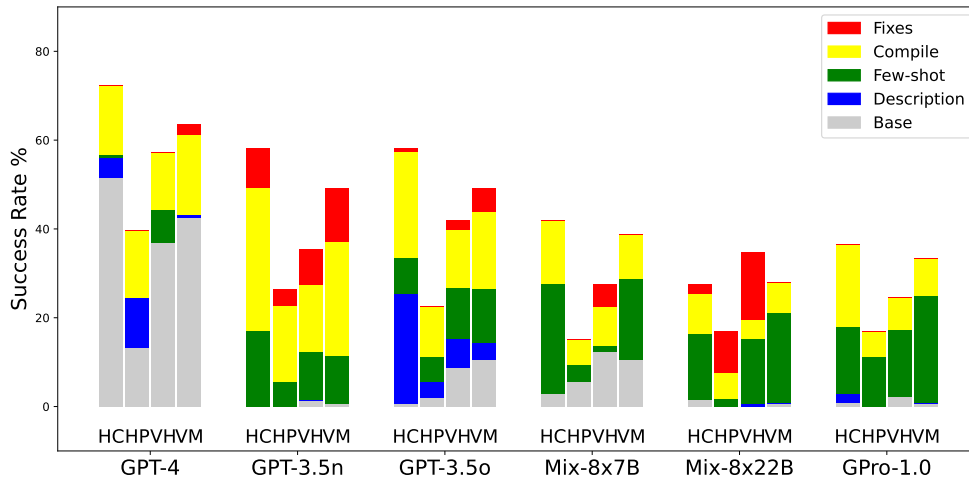


Figure 5.1: HDLAgent improves Chisel across all LLMs.

Chisel (Figure 5.1) is based on Scala; several LLMs know the basic syntax but only GPT-4 performs decently with Chisel. All the LLMs except GPT-4 had less than a 3% success rate with Chisel. Both "main context" (Description + Few-shot) and "compiler context" (Compile + Fixes) provide substantial benefit, demonstrating that all these components are necessary. Moreover, GPT-3.5o and GPT-3.5n perform better than high performing LLMs (GPT-4) without HDLAgent. HDLAgent is also able to improve

GPT-4, reaching a 69% success rate.

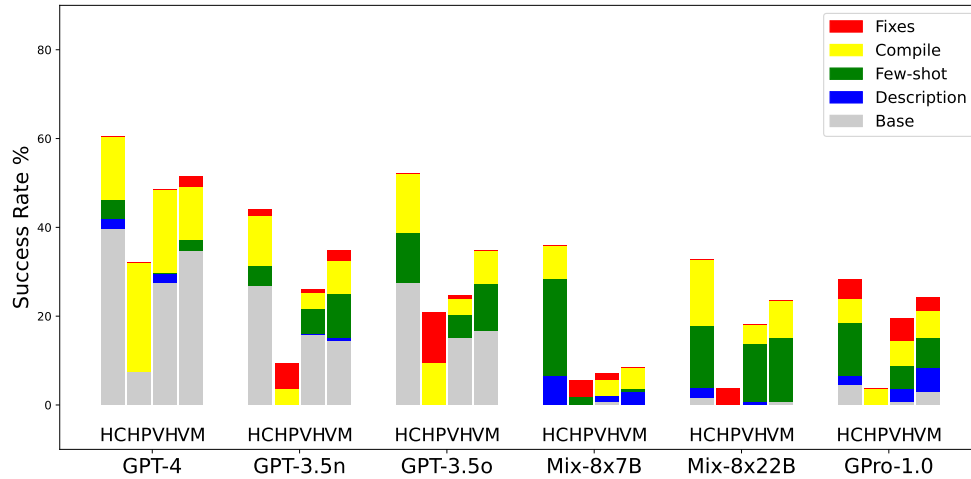


Figure 5.2: HDLAgent improves PyRTL across all LLMs.

PyRTL (Figure 5.2) is a Python-based HDL. OpenAI LLMs (GPT-4, GPT-3.5n, GPT-3.5o) can pass several tests without HDLAgent (Base); however, even these models have a low success rate, between 27% and 40% in PyRTL. When HDLAgent is enabled, the success rate increases to between 44% to 59%. Similar to the Chisel case, HDLAgent speedup comes from several factors.

As with Chisel, all the HDLAgent components are important but the "compiler context" (*Compile + Fixes*) is crucial for both PyRTL and Chisel. This is because these HDLs are Domain Specific Languages (DSLs), so the LLMs gets confused mixing the DSL hosts (Scala, Python) with their HDL-specific syntaxes. The compiler context is able to guide and solve the issues.

DSLX (Figure 5.3) is a Rust-like language. Since DSLX does not allow arbitrary pipelining, it cannot be evaluated against HDLEval-Pipe and it exhibits very poor per-

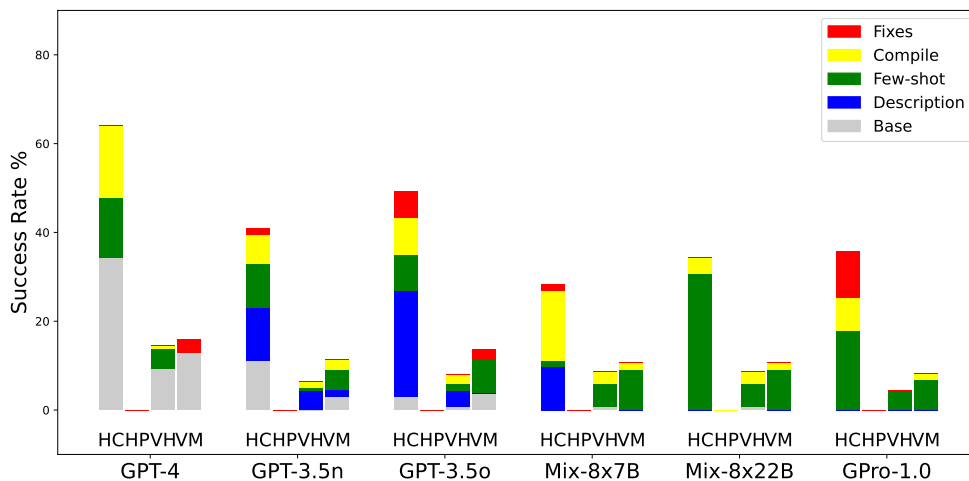


Figure 5.3: HDLAgent improves DSLX HDLEval-Comb across all LLMs.

formance with VerilogEval. GPT-4 has some knowledge of DSLX, and once again, HDLAgent significantly enhances the results across all LLMs. Unlike Chisel and PyRTL, DSLX is not a DSL. In this case, the "main context" (*HDL Description + Few-Shot*) is the biggest HDLAgent factor in this improvement, because explaining the Rust-like syntax and providing a few examples is more important than grounding the results with compile errors.

Verilog (Figure 5.4) has the best overall performance for the *Base*, which is when no HDLAgent is active. This is the expected behavior due to extensive training with Verilog syntax. It is also the only fair case where VerilogEval can be used to evaluate an HDL. Overall, HDLAgent has little impact on the models that are already proficient in Verilog; however, it has a significant impact on Mix-8x7B and Mix-8x22B, which have little Verilog knowledge. This is an interesting observation displaying how knowledge transfer can work even when the LLM is not familiar with Verilog.

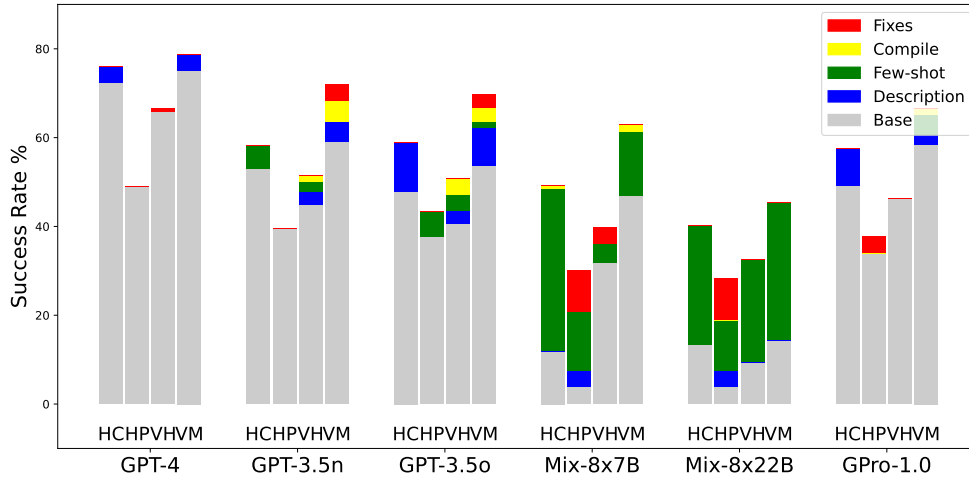


Figure 5.4: Verilog succeeds across benchmarks and LLMs .

One interesting observation regarding Verilog is that "compiler context" does not help recover from many errors. Part of the reason is that Verilog error messages use Yosys, which has error messages more cryptic than PyRTL (Python), Chisel (Scala), or DSLX. A stricter Verilog compiler with more readable errors could have a different distribution, but over 80% of the Verilog generated code in HDLEval-Comb compiles correctly at a first attempt, so there are limits on what compiler error grounding can provide.

When comparing LLMs, one unexpected behavior is that Mix-8x22B has lower performance than Mix-8x7B. The model was recently released, but we observe a high failure rate when it is attempting to follow directions. Using a "instruct" model will achieve better results. We kept the "base" model because it provides interesting insights on how different models improve and behave with HDLAgent.

One of the main HDLAgent objectives is to enable LLMs to use new HDLs.

When comparing different LLMs like GPT-3.5o and GPT-3.5n across various HDLs, trends indicate that the performance of HDLs remains relatively consistent regardless of the specific LLM used. For instance, with GPT-4, Verilog shows the highest success rate at 75%, which is close to its lowest for PyRTL with 59%. This pattern of performance is consistent across all tested LLMs. The worst performing LLM (Mix-8x22B) has a 53% Verilog success rate, while PyRTL the worst performing HDL Mix-8x22B has a 28% success rate. Without HDLAgent, many LLMs had a zero success rate.

5.1.2 HDLAgent Context Insights

This section discusses HDL Description and few-shot selection. One straightforward approach is to use the HDL reference manual directly. While this is feasible with GPT-4, Mix-8x7B, and GPro-1.0 due to their large context windows, its generally less effective than using a summarized HDL description. For example, using a full reference instead of a summary does not change results for GPro-1.0, but reduces success rate from 77% to 66% for GPT-4, and from 59% to 33% for Mix-8x7B. This indicates that future LLMs need to improve handling of length contexts, as all evaluated LLM struggle with this.

Figure 5.5 shows the DSLX, PyRTL, and Chisel success rate as different reference manuals are summarized for HDLAgent. Each bar shows a different LLM reference summarization prompt (Section ??) sorted by accuracy. The breakdown is the contribution of the few-shot examples and the HDL description. Interestingly, adding Few-shot always improves results, and removing HDL Description and just keeping few-shot exam-

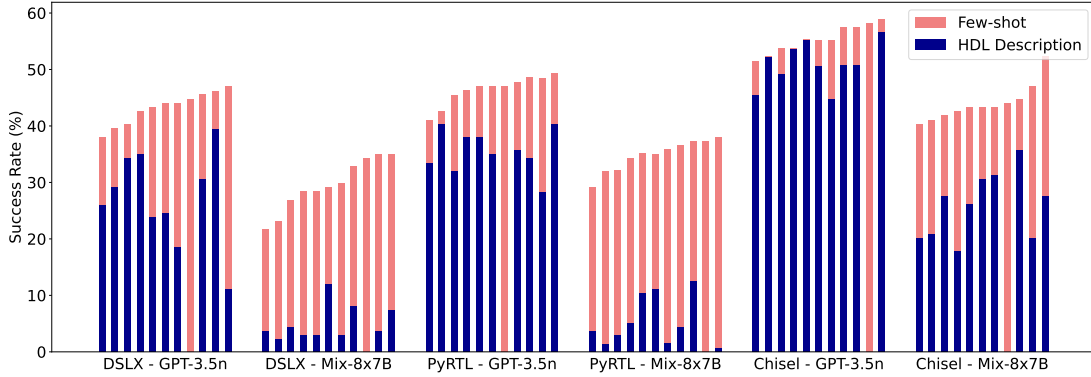


Figure 5.5: HDL description and few-shot help LLMs to improve results.

ples is a reasonable alternative. In some HDL/LLM combinations like Chisel/GPT-3.5n, using either Few-shot or Description works. For other combinations like DSLX/Mix-8x7B, HDL Description helps but Few-shot is necessary. Optimal results require both Few-shot and HDL Description.

5.1.3 Pass Sensitivity

Pass@k is a popular method that measures how results can be improved by generating multiple attempts. A k=5 means that when 5 LLM tries are used, at least one has the correct code generation. Table 5.1 shows tests passed for HDLEval-Comb for multiple LLMs and multiple pass@k values (1,5,10). Only the HDLEval-Comb results are shown, as it is the most representative benchmark for a multi-lang evaluation.

Less popular HDLs benefit more from higher pass@k values. For example, DSLX shows a 1.22 to 2.08 times improvement in test pass rates from pass@1 to pass@10. Verilog has between 1.16 and 1.45 times. This discrepancy is likely because the LLM,

		Verilog			Chisel			PyRTL			DSLX		
		k=1	k=5	k=10	k=1	k=5	k=10	k=1	k=5	k=10	k=1	k=5	k=10
GPT-4	Base	97	103	111	69	88	92	53	79	85	46	79	85
	HDLAgent	102	109	111	97	103	107	81	92	98	86	100	104
GPT-3.5n	Base	71	96	100	0	5	9	36	63	67	15	32	41
	HDLAgent	78	93	98	80	97	100	59	79	88	55	80	88
GPT-3.5o	Base	64	93	99	1	6	14	37	60	71	4	19	25
	HDLAgent	79	92	100	79	91	99	70	78	89	65	86	91
GPro-1.0	Base	66	97	105	1	5	12	6	17	31	0	0	0
	HDLAgent	77	96	99	49	84	88	38	66	77	48	74	82
Mix-8x7B	Base	16	39	50	4	12	17	0	1	2	0	0	0
	HDLAgent	66	86	95	60	80	86	48	71	82	38	72	79
Mix-8x22B	Base	18	65	78	2	12	18	2	8	13	0	0	6
	HDLAgent	72	96	101	35	79	89	39	67	72	47	75	81

Table 5.1: Pass@k results for HDLEval-Comb for different LLMs with just a Base query or with HDLAgent.

unfamiliar with the language, starts from an incorrect baseline and struggles to correct errors through compiler feedback. Not being able to fix is very rare in Verilog but over 10% of the DSLX tests have this problem. The higher the pass@k, the easier to avoid. Once the code compiles correctly, the failure rate for all the HDLs is comparable. This means that if a future HDLAgent improved the iterations or better selection point, it could further improve results.

Figure 5.6 provides more insights on the pass@k results. It shows the increase in accuracy as HDLAgent iterates with the compiler like GPro-1.0 across HDLs. We choose GPro-1.0 because it is one of the LLMs that need more iterations to converge. For Verilog, it converges very fast but for the other HDLs it needs 6 to 8 iterations to converge. More iterations do not improve, but changing the starting point like pass@5

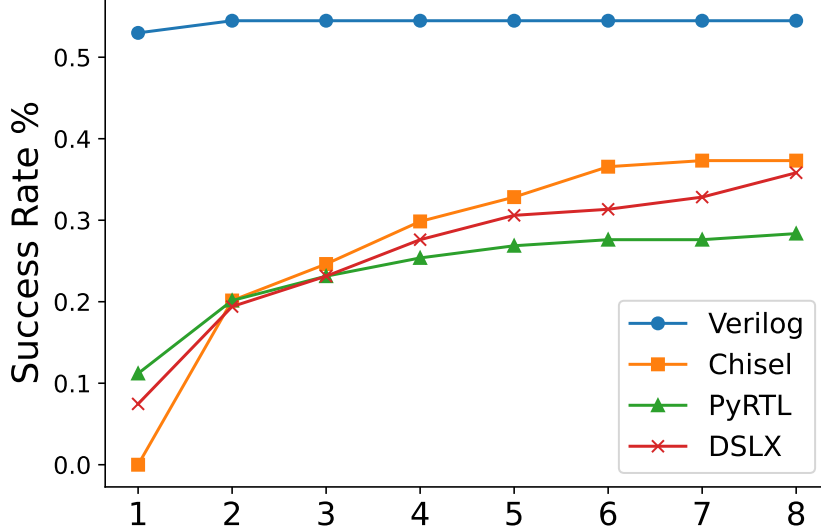


Figure 5.6: GPro-1.0 converges in a few iterations.

improves the results. Overall, 8 iterations is enough across languages because increasing iterations does not help in success rate but hurts in token usage.

When pass@5 and 8 iterations are used (Table 5.1), HDLAgent HDLs (Chisel, PyRTL, DSLX) perform better or equal to the same LLM with Verilog (Base). This is a main contribution on the paper showing that HDLAgent enables the use of small community HDLs.

5.1.4 Time and QoR

Execution time is an important metric for any AI Agent. Figure 5.7 shows the execution time boxplot for HDLEval-Comb with different LLMs, where both successful and failed tests are considered. All the languages besides Verilog go through a translation process to Verilog that adds overhead. In HDLAgent, the execution time is

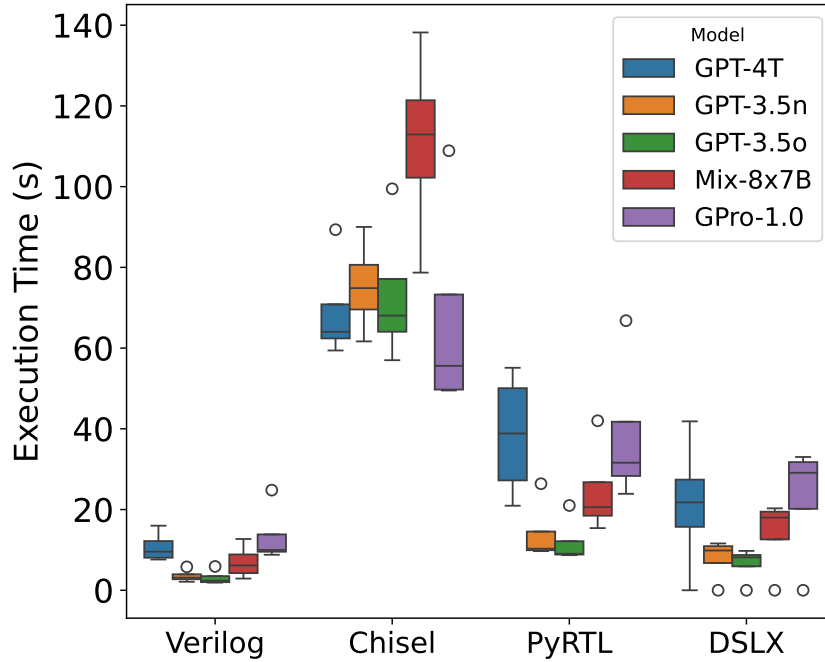


Figure 5.7: LLM and HDL affect total HDLAgent execution time.

a function of $\frac{\text{tokens}}{\text{second}}$, number of iterations, and external compiler speed.

Comparing across HDLs, the main outlier is Chisel. Around 2/3 of the execution time is spent in the FIRRTL compiler to generate Verilog. GPT-4 is faster because it has less errors and hence less iterations. PyRTL and DSLX are also slower than Verilog, but this is in part due to the additional iterations.

Comparing across LLMs, GPT-3.5n and GPT-3.5o tend to be faster overall as they combine less error iterations and speed of results. External $\frac{\text{tokens}}{\text{second}}$ benchmarking [1] points that GPro-1.0 is roughly 30% faster than GPT-3.5n and 4 times faster than GPT-4. HDLAgent results are different because of iterations and speed.

The Quality of Results (QoR) is crucial in hardware generation. Figure 5.8

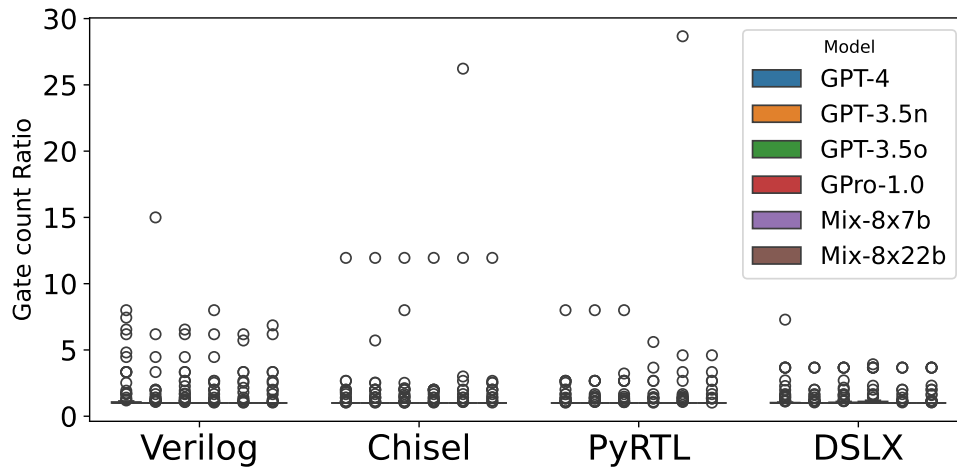


Figure 5.8: QoR is consistent across LLMs but different across HDLs.

shows the gate count ratio compared to the best implementation. A ratio of 1 indicates optimal gate count, while 2 indicates double the gate count. This figure only includes the successful runs using HDLAgent with HDLEval-Comb. The plot reveals significant Qor variation compared to the best implementation. Typically, the average varies due to one or two outliers. For instance, in PyRTL generated by Mix-8x7B, the average gate count is 1.63, but removing two outliers brings it down to 1.12. This means that the LLMs sometimes generate very inefficient code but it is not so frequent. A second observation suggests that GPT-4 may underperform; however, it successfully handles larger and more complex tests that affect the results. A third observation is that the code generated by various LLMs displays comparable efficiency. Among these, DSLX appears to be slightly more efficient. In DSLX generated by GPT-4, 80% of the generated code is the optimal with a 1 ratio. This seems to imply that an efficient compiler like XLS combined with a popular syntax can result in better results for generated HDL code.

5.1.5 Self-Reflection with Multi-Agent Debate

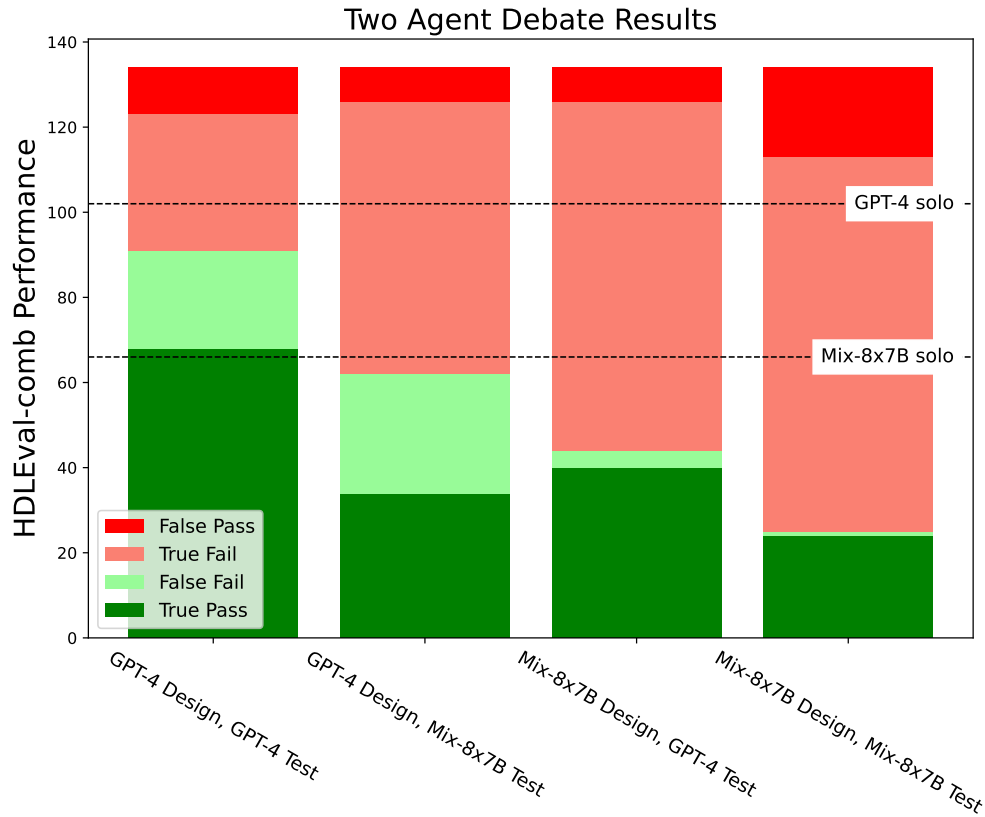


Figure 5.9: Introducing a testbench into the generative loop worsens results

As mentioned in Section ??, HDLAgent has an option to generate Verilog testbenches for the produced code. This section examines the results of passing the design through a testbench and iterating upon it in case of any testbench failures before attempting LEC. Figure 5.9 shows the results of using the best-performing and worst-performing LLMs, GPT-4 and Mix-8x7B respectively, as both designers and testers when writing Verilog for HDLEval-comb. If the initial version of the design failed its testbench, a second version would be generated and would be the one attempting

LEC. The results are compared to the horizontal dotted lines labeled "GPT-4 solo" and "Mix-8x7B solo," which represent the single Agent trials and implicitly the initial version design results.

There are four scenarios to consider:

Testbench passing and LEC passing ("True Pass"): This scenario indicates a test case reliably solved by the Agent, as the design complies with the testbench and passes LEC, showing that both Agents are well-trained on such problems.

Testbench failing but LEC passing ("False Fail"): Figure 5.9 shows that single Agent trials outperformed their double Agent counterparts in this category. This represents cases where the single Agent was able to solve the problem initially, indicating that the testbench was noncompliant with the prompt. The testbench and its feedback were ineffective in steering the designer Agent towards the correct solution, as the designer already had a strong grasp of how to implement the design. This suggests that LLMs are better designers than testers, as the testbench feedback led to no significant changes or incorrect changes in the design.

Testbench failing and LEC failing ("True Fail"): This scenario indicates that the LLMs could not write a proper testbench or design for the problem. The testbench logic does not match the design logic, implying either low confidence in the LLMs' answers or an inability to write quality testbenches. The results suggest that testbench feedback negatively affected the design, as each double Agent trial had failures in test cases that succeeded in the single Agent trials. This shows that the feedback from the testbench failure "infected" the initial design, causing the second iteration to fail

LEC when the initial design might have passed without the feedback. The regression may indicate that LLMs have difficulties interpreting testbenches and their feedback, rendering this double Agent process useless.

Testbench passing but LEC failing ("False Pass"): This scenario occurs when the problem is very difficult, and the LLMs do not understand the prompt or cannot solve it, yet they are confident in their answers and somehow write a testbench that passes the implementation. This may indicate that the tester LLM wrote a low-coverage testbench due to a poor understanding of the prompt. All these cases failed for both GPT-4 and Mix-8x7B in the single Agent trials, further supporting this observation.

Overall, the results suggest that HDLAgent's current multi-Agent debate scheme provides no advantage for RTL design. Some tests that would have otherwise passed fail due to the feedback from failed testbenches. Developing a more cohesive arbitration scheme for inter-Agent debate, as well as fine-tuning for testbench generation and interpretation, is left for future work.

5.1.6 Insights for HDLs at the age of LLMs

The goal of this section is to show some shortcomings in HDLs that need to be addressed to improve accuracy in an LLM world.

5.1.6.1 Verilog

Verilog is the language that LLMs know the best. For top-performing LLMs like GPT-4, the main challenge lies in handling pipelining. Verilog allows for unre-

stricted pipelining, which deviates from the traditional Von Neumann architecture or non-hardware program structure. GPT-4 effectively generates combinational logic because a typical program without recursion or memory access can be directly translated to Verilog or combinational logic. Improving pipelining remains an open research question that needs to be addressed to enhance LLMs' performance hardware tasks.

5.1.6.2 Chisel

Besides the common pipelining issue, Chisel LLM code generation needs help to interface Chisel code to Verilog. As a part of compilation process, the Verilog generated module's IO appends "io_" to all names. Additional clock and reset signals are created by default, even if unused in the original Chisel code. Listing 1 shows the resulting Verilog from a compiled Chisel implementation of a full adder circuit.

To interface modules, HDLAgent adjusts the IO to perform testing. Postprocessing is used to remove the unused signals as well as renaming those modified to their originals to match the circuit specification. This is necessary as the first step of the LEC checks that the two modules' IOs match, otherwise a truthful comparison is impossible and the LEC fails.

In addition, Chisel shares a problem with PyRTL of being a Domain Specific Language (DSL), and the LLMs use incorrect syntax.

```

module full_adder(
  input  clock,
  input  reset,
  input  io_a,
  input  io_b,
  input  io_cin,
  output io_sum,
  output io_cout
);

```

Listing 1: Chisel IOs have name changes.

5.1.6.3 PyRTL

PyRTL shares common problems with Verilog and Chisel, but it also has a problem with semantics.

A DSL problem is when the LLM generates Python syntax to implement logic instead of the PyRTL syntax. Listing 2 invalid cases uses Python "inp 1" instead of the PyRTL shift right logical library call. Many of those problems generate errors which are caught by HDLAgent, and it is able to fix with further HDLAgent iterations.

```

inp = pyrtl.inpput(4, 'inp')
out = pyrtl.Output(4, 'out')
out <<= inp ^ pyrtl.shift_right_logical(inp, 1)
# equivalent: out <<= pyrtl.concat(inp[3] ^ 0, inp[3] ^ inp[2], inp
  [2] ^ inp[1], inp[1] ^ inp[0])
# CORRECT   : out <<= pyrtl.concat(inp[3]      , inp[3] ^ inp[2], inp
  [2] ^ inp[1], inp[1] ^ inp[0])
# INVALID   : out <<= inp ^ (inp>>1) # Invalid, >> is a python shift
  not PyRTL

```

Listing 2: PyRTL issues generating right shift.

Besides DSL problems, PyRTL has errors due to inconsistent semantics. In Verilog and Chisel, a right shift logical of a positive number reduces the size. For example if inp has 4 bits, and it is right shifted once, the output has 3 bits. Not in

PyRTL, it has 4 bits but the most significant bit is a zero. Listing 2 showcases the problem in one HDLEval test. The most significant bit is xored with zero which is not the expected result.

5.1.6.4 DSLX

```
fn add_7_to_11() -> Outputs {
  //add values from 7 to 11 (exclusive)
  let base = u16:7;
  let res = for (i, accum): (u16, u16) in u16:0..u16:4 {
    accum + base + i
  }(u16:0);
  Outputs { result: res }
}
```

Listing 3: Rust DSLX special loop syntax.

DSLX presented a different set of challenges compared to DSLs like Chisel and PyRTL. Since it does not support unrestricted pipelining, only combinational logic is used in this section's feedback.

DSLX shares input/output generation issues with Chisel and PyRTL but faces even greater challenges. DSLX has a single unnamed output for functions named "out" during Verilog generation. DSLX solution to multiple outputs is to use a struct. HDLAgent addresses it by post-processing the generated Verilog and modifying the output port name to match the desired name. A better solution that requires DSLX semantic changes would be to adopt a Go-like syntax that allows for multiple named outputs and ensures Verilog generation respects those outputs.

Another interesting source of errors stems from DSLX being "similar to Rust".

If the HDLAgent’s HDL Description mentions that “DSLX is similar to Rust...” it generates even more errors. Even without this sentence, the LLM sometimes generates legal Rust but illegal DSLX code. Some differences are easy to spot, such as DSLX’s “assert(cond)” versus Rust’s “assert_eq!(cond),” while others, like the presence of Rust annotations like “[test]” in DSLX code, are more subtle. To address the “similar but not the same” syntax issues, it is suggested to avoid mentioning the similarity and catch any discrepancies during compilation time, generating a compile error for HDLAgent to fix.

A more complicated case involves semantic changes. Since DSLX cannot describe circuits with mutable variables, its expressions cannot describe state changes over a loop, making it incompatible with the Rust loop semantics. Instead, these expressions have an accumulator value separate from the iterator, creating a return value calculated by the body of the for loop. As shown in Listing 3, the for loop body sums the values between 7 and 11 by accumulating the base value of 7 and the iterator value in the range of 0 to 4 each loop “iteration.” This deviation from standard loop semantics required a dedicated code snippet and explanation in both the initial and supplemental contexts to correct the LLM’s often incorrect assumptions about DSLX’s generative for loop syntax. Addressing these changes will help LLMs to perform better with less HDLAgent iterations.

Chapter 6

Future Work and Conclusions

6.1 Future Work and Conclusions

Large Language Models (LLMs) have the potential to revolutionize computer science. This paper introduces HDLAgent, an AI agent that significantly enhances LLMs' HDL code generation for less popular HDLs like Chisel, PyRTL, and DSLX. Supporting multiple HDLs without LLM tuning is crucial for pioneering new HDLs that leverage LLM capabilities. The paper presents several challenges and recommendations for existing and future HDLs and AI agents for chip design.

We also present HDLEval, a test suite for assessing HDL compatibility with LLMs or AI agents. This benchmark allows various HDLs to be evaluated and provides a new set of challenging problems based on competitions, highlighting pipelining versus combinational performance issues that the AI agent community should address.

Evaluation reveals that HDLAgent enables emerging HDLs to improve across

all LLMs. The best performing LLM is GPT-4, which had a 72% success rate for Verilog and 34% for DSLX without HDLAgent. Once HDLAgent is applied, all the HDLs achieve between 72% and 82% in pass@10. This indicates that HDLAgent can facilitate knowledge transfer across HDLs and boost performance beyond the GPT-4 Verilog baseline. This property is consistent across all LLMs, with the performance of all HDLs with HDLAgent always surpassing the Verilog performance when HDLAgent is not applied in pass@10.

The evaluations of HDLAgent and HDLEval in this paper highlight the challenges faced by the AI agent community in chip design. These include addressing QoR outliers, reducing execution time and token count, overcoming difficulties with DSLs, improving compile error messages to guide LLMs, managing large outputs, and enhancing pipelining.

In summary, HDLAgent and HDLEval enable the use of LLMs for HDLs beyond Verilog. The code for HDLAgent and HDLEval will be open-sourced to further benefit the community, providing several insights and addressing various challenges.

Bibliography

- [1] Artificial Analysis. <https://artificialanalysis.ai/models/gpt-35-turbo>. Online; accessed on April 2024.
- [2] Toufique Ahmed and Premkumar Devanbu. Few-shot training llms for project-specific code-summarization. *arXiv preprint arXiv:2207.04237*, 2022.
- [3] 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.
- [4] 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.
- [5] Xinyun Chen, Maxwell Lin, Nathanael Schärli, and Denny Zhou. Teaching large language models to self-debug, 2023.
- [6] John Clow, Georgios Tzimpragos, Deeksha Dangwal, Sammy Guo, Joseph McMahhan, and Timothy Sherwood. A pythonic approach for rapid hardware prototyping and instrumentation. In *Field Programmable Logic and Applications (FPL), 2017 27th International Conference on*, pages 1–7. IEEE, 2017.
- [7] Yihong Dong, Xue Jiang, Zhi Jin, and Ge Li. Self-collaboration code generation via chatgpt, 2023.
- [8] efabless. Efabless 1st competition winners. <https://efabless.com/genai/challenges/1>, 2023.
- [9] efabless. Efabless 2nd competition winners. <https://efabless.com/genai/challenges/2-winners>, 2023.
- [10] Zhiyu Fan, Xiang Gao, Abhik Roychoudhury, and Shin Hwei Tan. Improving

- automatically generated code from codex via automated program repair. *arXiv preprint arXiv:2205.10583*, 2022.
- [11] Google. XLS Website. <https://github.com/google/xls/>, 2022.
- [12] Xue Jiang, Yihong Dong, Lecheng Wang, Zheng Fang, Qiwei Shang, Ge Li, Zhi Jin, and Wenpin Jiao. Self-planning code generation with large language models, 2023.
- [13] Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen tau Yih, Tim Rocktäschel, Sebastian Riedel, and Douwe Kiela. Retrieval-augmented generation for knowledge-intensive nlp tasks, 2021.
- [14] Bill Yuchen Lin, Abhilasha Ravichander, Ximing Lu, Nouha Dziri, Melanie Sclar, Khyathi Chandu, Chandra Bhagavatula, and Yejin Choi. The unlocking spell on base llms: Rethinking alignment via in-context learning, 2023.
- [15] Haokun Liu, Derek Tam, Mohammed Muqeeth, Jay Mohta, Tenghao Huang, Mohit Bansal, and Colin A Raffel. Few-shot parameter-efficient fine-tuning is better and cheaper than in-context learning. *Advances in Neural Information Processing Systems*, 35:1950–1965, 2022.
- [16] Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation, 2023.

- [17] Mingjie Liu, Nathaniel Pinckney, Brucek Khailany, and Haoxing Ren. Verilog-eval: Evaluating large language models for verilog code generation. *arXiv preprint arXiv:2309.07544*, 2023.
- [18] Shang Liu, Wenji Fang, Yao Lu, Qijun Zhang, Hongce Zhang, and Zhiyao Xie. Rtlcoder: Outperforming gpt-3.5 in design rtl generation with our open-source dataset and lightweight solution, 2024.
- [19] Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegrefe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, Shashank Gupta, Bodhisattwa Prasad Majumder, Katherine Hermann, Sean Welleck, Amir Yazdanbakhsh, and Peter Clark. Self-refine: Iterative refinement with self-feedback, 2023.
- [20] Seungjun Moon, Yongho Song, Hyungjoo Chae, Dongjin Kang, Taeyoon Kwon, Kai Tzu iunn Ong, Seung won Hwang, and Jinyoung Yeo. Coffee: Boost your code llms by fixing bugs with feedback, 2023.
- [21] Ansong Ni, Srini Iyer, Dragomir Radev, Ves Stoyanov, Wen-tau Yih, Sida I Wang, and Xi Victoria Lin. Lever: Learning to verify language-to-code generation with execution. *arXiv preprint arXiv:2302.08468*, 2023.
- [22] Theo X. Olausson, Jeevana Priya Inala, Chenglong Wang, Jianfeng Gao, and Armando Solar-Lezama. Is self-repair a silver bullet for code generation?, 2023.
- [23] Florian Tambon, Arghavan Moradi Dakhel, Amin Nikanjam, Foutse Khomh,

- Michel C. Desmarais, and Giuliano Antoniol. Bugs in large language models generated code: An empirical study, 2024.
- [24] 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 and Test in Europe Conference and Exhibition (DATE)*, pages 1–6, 2023.
- [25] Shailja Thakur, Baleegh Ahmad, Hammond Pearce, Benjamin Tan, Brendan Dolan-Gavitt, Ramesh Karri, and Siddharth Garg. Verigen: A large language model for verilog code generation, 2023.
- [26] Shailja Thakur, Jason Blocklove, Hammond Pearce, Benjamin Tan, Siddharth Garg, and Ramesh Karri. Autochip: Automating hdl generation using llm feedback, 2023.
- [27] Yun-Da Tsai, Mingjie Liu, and Haoxing Ren. Rtlfixer: Automatically fixing rtl syntax errors with large language models, 2024.
- [28] Hanbin Wang, Zhenghao Liu, Shuo Wang, Ganqu Cui, Ning Ding, Zhiyuan Liu, and Ge Yu. Intervenor: Prompt the coding ability of large language models with the interactive chain of repairing, 2023.
- [29] Zejun Wang, Jia Li, Ge Li, and Zhi Jin. Chatcoder: Chat-based refine requirement improves llms’ code generation, 2023.
- [30] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia,

- Ed Chi, Quoc Le, and Denny Zhou. Chain-of-thought prompting elicits reasoning in large language models, 2023.
- [31] Clifford Wolf. Yosys Open SYnthesis Suite. <https://github.com/YosysHQ/yosys>, 2022. Online; accessed on December 2022.
- [32] Chunqiu Steven Xia and Lingming Zhang. Conversational automated program repair. *arXiv preprint arXiv:2301.13246*, 2023.
- [33] Chengrun Yang, Xuezhi Wang, Yifeng Lu, Hanxiao Liu, Quoc V. Le, Denny Zhou, and Xinyun Chen. Large language models as optimizers, 2023.
- [34] Kaiyuan Yang, Haotian Liu, Yuqin Zhao, and Tiantai Deng. A new design approach of hardware implementation through natural language entry. *IET Collaborative Intelligent Manufacturing*, 5(4):e12087, 2023.
- [35] Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. React: Synergizing reasoning and acting in language models, 2023.
- [36] 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.
- [37] Da Yin, Faeze Brahman, Abhilasha Ravichander, Khyathi Chandu, Kai-Wei Chang, Yejin Choi, and Bill Yuchen Lin. Lumos: Learning agents with unified data, modular design, and open-source llms, 2023.

- [38] 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.
- [39] Qinkai Zheng, Xiao Xia, Xu Zou, Yuxiao Dong, Shan Wang, Yufei Xue, Zihan Wang, Lei Shen, Andi Wang, Yang Li, Teng Su, Zhilin Yang, and Jie Tang. Codegeex: A pre-trained model for code generation with multilingual evaluations on humaneval-x, 2023.
- [40] Li Zhong and Zilong Wang. Can chatgpt replace stackoverflow? a study on robustness and reliability of large language model code generation, 2024.
- [41] Wangchunshu Zhou, Yuchen Eleanor Jiang, Long Li, Jialong Wu, Tiannan Wang, Shi Qiu, Jintian Zhang, Jing Chen, Ruipu Wu, Shuai Wang, Shiding Zhu, Jiyu Chen, Wentao Zhang, Ningyu Zhang, Huajun Chen, Peng Cui, and Mrinmaya Sachan. Agents: An open-source framework for autonomous language agents, 2023.