VerilogCoder: Autonomous Verilog Coding Agents with Graph-based Planning and Abstract Syntax Tree (AST)-based Waveform Tracing Tool

Chia-Tung Ho, Haoxing Ren, Brucek Khailany

NVIDIA Research
chiatungh@nvidia.com, haoxingr@nvidia.com, bkhailany@nvidia.com

Abstract

Due to the growing complexity of modern Integrated Circuits (ICs), automating hardware design can prevent a significant amount of human error from the engineering process and result in less errors. Verilog is a popular hardware description language for designing and modeling digital systems; thus, Verilog generation is one of the emerging areas of research to facilitate the design process. In this work, we propose VerilogCoder, a system of multiple Artificial Intelligence (AI) agents for Verilog code generation, to autonomously write Verilog code and fix syntax and functional errors using collaborative Verilog tools (i.e., syntax checker, simulator, and waveform tracer). Firstly, we propose a task planner that utilizes a novel Task and Circuit Relation Graph retrieval method to construct a holistic plan based on module descriptions. To debug and fix functional errors, we develop a novel and efficient abstract syntax tree (AST)-based waveform tracing tool, which is integrated within the autonomous Verilog completion flow. The proposed methodology successfully generates 94.2% syntactically and functionally correct Verilog code, surpassing the state-of-the-art methods by 33.9% on the VerilogEval-Human v2 benchmark.

Code — https://github.com/NVlabs/VerilogCoder

Introduction

Designing modern integrated circuits requires designers to write code in hardware description languages such as Verilog and VHDL to specify hardware architectures and model the behaviors of digital systems. Due to the growing complexity of VLSI design, writing Verilog and VHDL is time-consuming and prone to bugs, necessitating multiple iterations for debugging functional correctness. Consequently, reducing design costs and designer effort for completing hardware specifications has emerged as a critical need.

Large Language Models (LLMs) have shown remarkable capacity to comprehend and generate natural language at a massive scale, leading to many potential applications and benefits across various domains. In the field of coding, LLM can assist developers by suggesting code snippets, offering solutions to fix bugs, and even generating the code with explanation (Mastropaolo et al. 2023; Nijkamp et al. 2023). Several works have focused on refining LLMs with selected datasets for Verilog generation (Liu et al. 2023a; Thakur et al. 2024). Pei *et al.* (Pei et al. 2024) proposed leveraging instruct-tuned LLM and a generative discriminators to optimize Verilog implementation with the considerations of PPA (Power, Performance, Area). However, these works lack of a mechanism to fix syntactic or functional errors, thus, they still struggle to generate functionally correct Verilog code. Recently, Tsai *et al.* (Tsai, Liu, and Ren 2023) presented an autonomous agent framework incorporating feedback from simulators and Retrieval Augmented Generation to fix syntax errors, but it failed to improve the functional success rate.

In this work, we propose a framework leveraging multiple Artificial Intelligence (AI) agents for Verilog code generation, which autonomously writes the Verilog code and fixes syntax and functional errors using collaborative Verilog toolkits and the ReAct (Yao et al. 2022) technique. In the framework, we develop a novel task planner to generate high-quality plans, and integrate a crafted Abstract Syntax Tree (AST)-based waveform tracing tool for improving the functional success rate. Our contributions are as follows.

- We are the first to explore the use of mult-AI agents for autonomous Verilog code completion, including syntax correction, and functional correction.
- We have developed a novel Task and Circuit Relation Graph (TCRG) based task planner to create a high-quality plan with step-by-step sub-tasks and related circuit information (i.e., signal, signal transition, and single examples).
- We propose a novel Abstract Syntax Tree (AST)-based waveform tracing tool to assist the LLM agent in fixing functional correctness.
- We conduct extensive and holistic ablation studies of each key component on the VerilogEval-Human v2 benchmark (Pinckney et al. 2024). We demonstrate the proposed VerilogCoder achieve 94.2% pass rate, including syntax and functional correctness, and outperform the one of the state-of-the-art methods by 33.9%.

The remaining sections are organized as follows. We first review prior works on AI agents and multi-AI agent systems. Then, we introduce and describe our novel VerilogCoder in details. Lastly, we present main experimental results and conclude the paper.

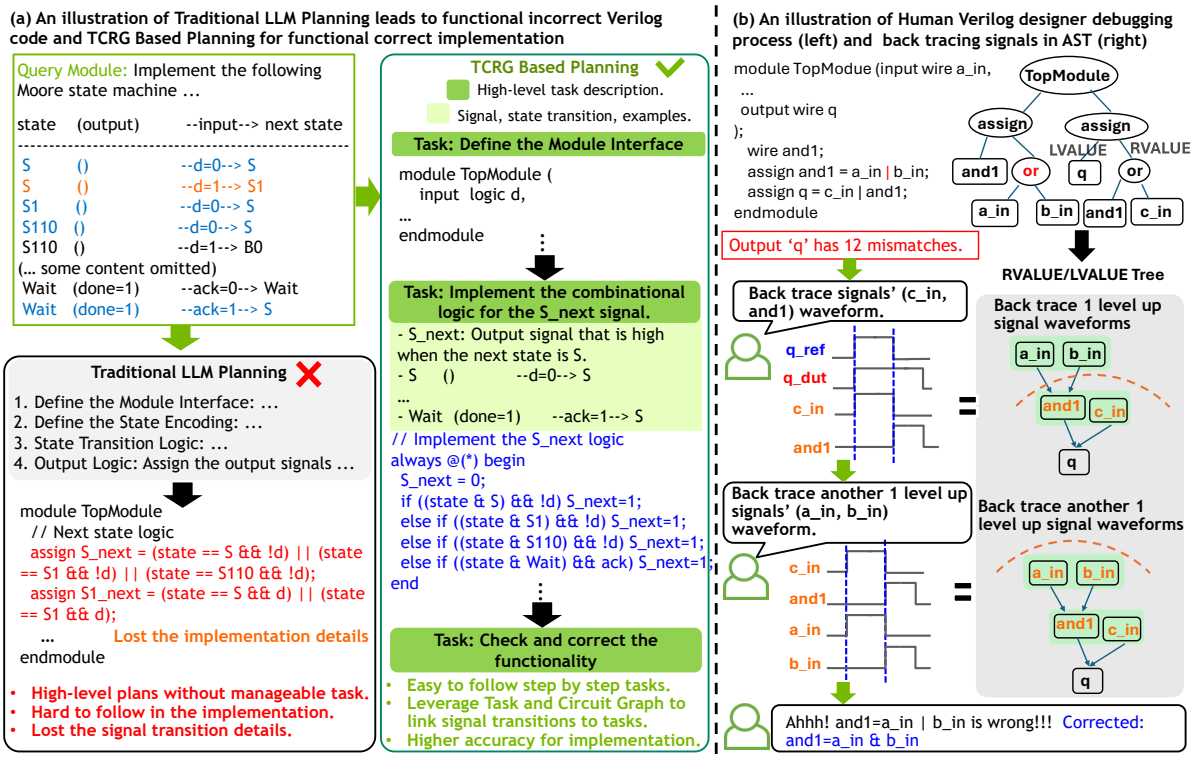

Figure 1: Illustrations of (a) traditional LLM planning versus TCRG based planning, and (b) human Verilog designer debugging process and AST signal back tracing in Motivation and Preliminary Study section.

Background

Autonomous agents have long been a research focus in academic and industrial communities across various fields. Recently, LLMs have shown great potential of human-level intelligence through the acquisition of vast amounts of knowledge, documents and textbooks, leading to a surge in research on LLM-based autonomous agents. Here, we firstly review prior AI agent works and introduce the multi-AI agent frameworks below.

AI Agent

Several works study the architecture of LLM-based autonomous agents to effectively perform diverse tasks (Wang et al. 2024; Weng 2023). From these studies, an LLM-powered autonomous agent system is composed of several key components: (a) Planning, (b) Memory, (c) Action, etc. The planning module enables the agent to break down large tasks into smaller, manageable sub plans, enabling efficient handling of complex tasks. In the memory module, short-term memory consists of chat history and in-context learning techniques to guide LLM actions. Long-term memory consolidates important information over time and provides the agent with the capability to retain and recall it over extended periods. The action module translates the agent's decisions into outcomes for solving tasks. The actions of an autonomous LLM-based agent can be categorized into two classes: (1) External tools for additional information and the expansion of the agent's capabilities, and (2) Internal knowledge of the LLMs, such as summary, conversation, etc.

Recently, AI agents empowered by LLMs (i.e., OpenDevin (OpenDevin Team 2024), SWE-agent (Yang et al. 2024), AgentCoder (Huang et al. 2023), etc) have shown impressive performance in software engineering for solving real world challenging benchmarks (i.e., SWE-Bench, HumanEval) through planning, memory management, actions involving external environment tools.

Multi-AI Agents

In addition to single AI agents, many researchers are starting to explore the capabilities of multiple AI agents for solving complex tasks. Autogen (Wu et al. 2023) has been proposed to enable multiple agents to operate in various modes (i.e., hierarchical chat, multi-agent conversation, etc.) that employ combinations of LLMs, human inputs, and tools. crewAI (crewAI Inc. 2024) facilitates process-oriented solving with a crew of customized multi-AI agents operating as a cohesive unit. Currently, the applications of these multi-AI agent frameworks are mostly for general tasks (i.e., QA, summarization, coding copilot, etc.).

However, these agent frameworks cannot be directly used for designing hardware because solving hardware tasks requires integrated domain knowledge and specific hardware design toolkits (i.e., circuit simulators, waveform debugging tools) to analyze signals, trace signal transitions, and decompose tasks into manageable sub-tasks from circuit architecture and signal transaction perspectives.

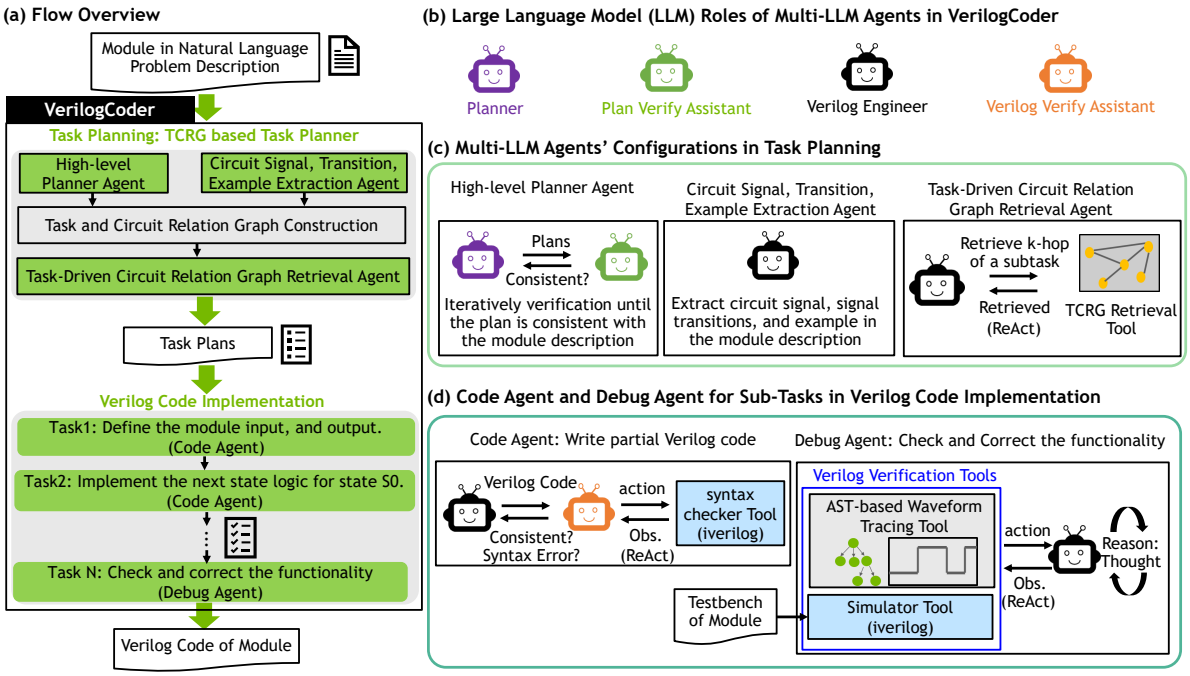

Figure 2: Flow overview of VerilogCoder. (a) Overall flow for Verilog code implementation task. (b) LLM roles of multi-LLM agents. (c) Multi-LLM agents in Task Planning. (d) Multi-LLM Agents for sub-tasks in Verilog Code Implementation.

Motivation and Preliminary Study

Given a hardware module description, hardware designers usually write Verilog using the following steps: (1) decompose the task into manageable sub-tasks, (2) implement Verilog code for each sub-task, and (3) iterate between Verilog simulations, signal waveform debugging, and code updates until all output signals match expected behavior. It is very challenging to autonomously complete a functionally correct Verilog module using LLM agents since it requires domain knowledge to break down the task into meaningful subtasks and comprehend the hardware descriptions and waveform during the functional debug process. Consequently, we first discuss the issues of using traditional LLM planning on writing Verilog code of a Finite State Machine (FSM) module. Then, we study the functional debug process of a Verilog module and propose a debugging tool that enables LLM agents to autonomously correct the functional errors.

Planning

Planning is one of the core modules for an agent (Wang et al. 2024; Weng 2023) to decompose a complex task into manageable sub-tasks. For Verilog coding, the traditional LLM-generated plans usually lack of the details of relevant signals, and signal transitions for each sub-task, thus, leading to incorrect functionality implementation of Verilog modules. Figure 1(a) shows an illustration of using the traditional LLM and TCRG based planning methods on a FSM module. The implementation of traditional LLM planning lost part of the state transitions for S_next, and S1_next signals, thus, leading to an incorrect FSM module. Therefore, it is important to guide the agent to implement each sub-task step by step with essential signals, and state transition information. Once the state transition information and signal definitions are included with the sub-task plan, LLM can generate the correct code. Signals and state transition information can be extracted from the problem descriptions. In this work, we structure sub-task, signal, and state transition information in a graph format and call it the TCRG. Consequently, we study the benefits of leveraging the TCRG to assist the planning to generate sub-tasks that include not only high-level task goals but also the signal, and signal transition information to complete functional correct Verilog module.

Functional Debug with Waveform

Figure 1(b) shows a typical functional debug process for a human Verilog designer. Given the mismatched signals, a human Verilog designer traces the signals and their waveform iteratively until they know how to fix the functionality. This backtracing procedure is the same as tracing the RVALUE of the target signals in the AST. Inspired by the human Verilog designer debug process, we propose to incorporate the hardware signal structure, and waveform, to assist LLM agents in fixing functional errors of the generated Verilog module. This process can be implemented with a tool based on AST and waveform tracing. Several prior works (Alon et al. 2019; Bairi et al. 2024; Bui et al. 2023) developed AST-based methods/tools (i.e., encoded AST paths, AST dependency graph, etc) to assist LLM in capturing structural information from the code for improving the capabilities of LLMs on various software engineering tasks such as code classification, understanding, and code completion. Here, the use of AST for signal tracing in our work is novel.

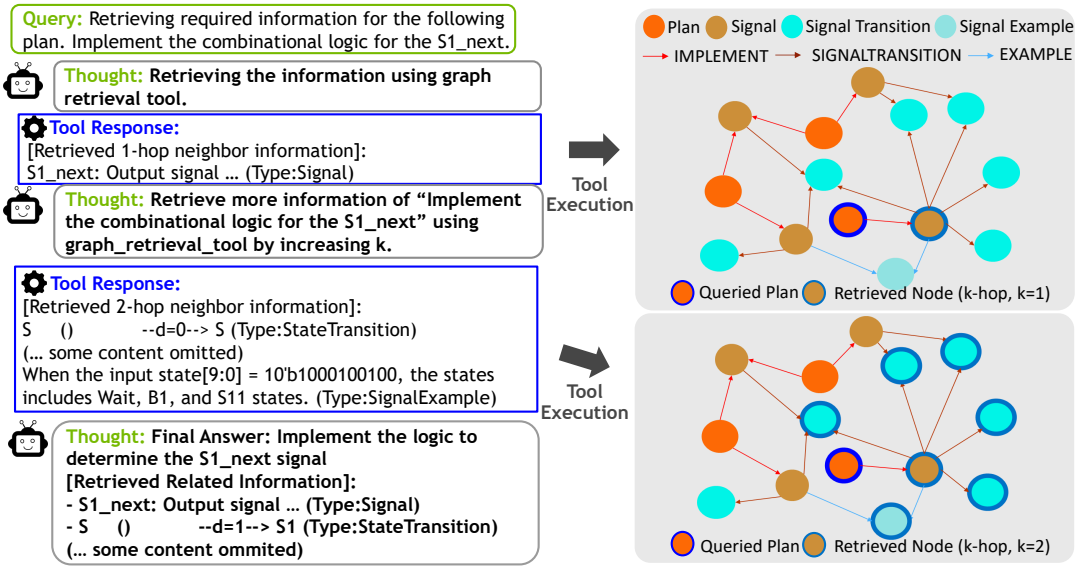

Figure 3: An illustration of task-driven circuit relation graph retrieval agent reasoning and interacting with the developed *TCRG retrieval tool* to enrich the task with the relevant circuit and signal descriptions.

VerilogCoder

We introduce the details of VerilogCoder, which consist of a task planning and Verilog code implementation. The multi-AI agents of VerilogCoder operate with developed TCRG retrieval and Verilog tools through the ReAct (Yao et al. 2022) technique in a cohesive and orchestrated manner.

Flow Overview

We outline the overall flow of VerilogCoder in Figure 2(a). Given the natural language problem description of a module (Pinckney et al. 2024), the novel Task and Circuit Relation Graph (TCRG) based task planner first generates the task plans. Then, a task dependency graph is built according to the task plans and its sub-tasks are assigned to Multi-LLM agents that write Verilog code and correct the functionality using a collaborative Verilog toolkit (i.e., syntax checker, simulator, and the proposed novel AST-based waveform tracing tool). In the flow, each agent may consist of multiple LLMs with different roles, which are listed in Figure 2(b), to complete each step correctly and consistently. Some of the agents are equipped with provided TCRG and Verilog tools to reason and act through Thought, Action, and Observation tracing of the ReAct prompting mechanism (Yao et al. 2022). For agent memory, we keep the original query and the last four chats in the chat history. The corresponding testbench of the module is used only for running Verilog simulator to check the functional correctness.

Task Planning

We introduce a novel and effective TCRG based Task Planner that constructs a high-quality plan encompassing not only the high-level objectives but also the relevant descriptions or definitions of signals, signal transitions, and examples for each sub-task. Recently, many works have utilized large language models (LLMs) to analyze texts and extract entities and relations for knowledge graph construction (Edge et al. 2024; Kommineni, König-Ries, and Samuel 2024; Zhang and Soh 2024). Inspired by these works, we leverage LLM agents to construct the TCRG with designer guidelines. In Figure 2(a), the task plan generation flow comprises four components: (1) High-level planner agent, (2) Circuit signal, transition, example extraction agent, (3) TCRG construction, and (4) Task-driven circuit relation graph retrieval agent. Figure 2(c) shows the configuration and tools of each AI agent in TCRG based Task Planner.

High-level planner agent The high-level planner agent consists of a planner and a plan verification assistant, as shown in Figure 2(c). Given the module description or specification, the planner first decomposes the task into sub-tasks, which mostly consist of high-level task descriptions. Then, the plan verification assistant checks the consistency between the sub-tasks and the module description, providing suggestions to modify the plan if any inconsistencies are found. This iterative process continues until the planner's plan is verified to be consistent with the module description.

Circuit signal, transition, example extraction agent A LLM acts as a Verilog engineer, extracting circuit signals, transitions, and examples from the given module description or specification into JSON format, as shown in Figure 2(c). The extracted information is represented as nodes in the subsequent TCRG construction. The examples of extracted signals, transitions, and signal examples are "w: input signal examined by FSM in state B", "State A to State B: FSM moves to state B when s = 1.", and "For example, when the input w = 1, 1, 0 in these three clock cycles, output z is set to 1 for the following cycle.", respectively.

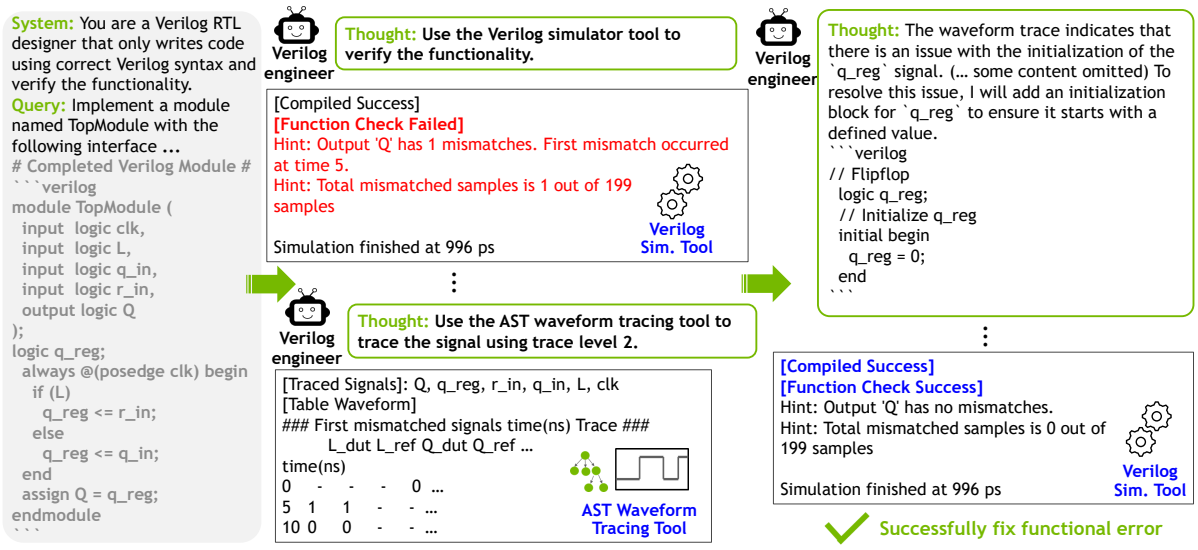

Figure 4: An example of Debug Agent reasoning and interacting with simulator and AST-based waveform tracing tool.

TCRG construction We create nodes from the previously generated high-level task descriptions, extracted circuit signals, transitions, and examples. We then sequentially create the relations (edges) between nodes: task nodes to signal nodes, signal nodes to transition nodes, and signal nodes to example nodes, using "IMPLEMENTS", "SIGNALTRANSITION", and "EXAMPLES" relationships, respectively.

Task-driven circuit relation graph retrieval agent Here, an LLM (acting as a Verilog Engineer) autonomously retrieves relevant signal and circuit descriptions and compiles this information for each sub-task using the collaborative *TCRG retrieval tool* through Thought-Action-Observation ReAct tracing (Yao et al. 2022), as shown in Figure 2(c). We firstly introduce the tool and then describe the workflow of the retrieval agent.

TCRG retrieval tool assists the task-driven circuit relation graph retrieval agent in obtaining relevant descriptions or definitions of signals, signal transitions, and examples related to a specified sub-task in the constructed TCRG. The inputs are the sub-task description in string format and an integer value, k, which indicates the number of hops for retrieval from the sub-task node in the graph. Here, k is determined by the AI agent automatically through the Thought-Action-Observation reasoning trace. The output consists of the retrieved k-hop signals, signal transitions, and examples corresponding to the sub-task node.

The retrieval agent reasons and interacts with the *TCRG retrieval tool* to incorporate additional information as illustrated in Figure 3. Ultimately, the retrieval agent compiles the retrieved circuit and signal information from the graph and removes irrelevant information from the final answer.

Verilog Code Implementation

We describe the Verilog code implementation flow of writing Verilog code and ensuring the functionality of the written Verilog module in detail. Given a task plan, the task dependency graph is created. A child task can not be executed until all its parent tasks have been completed without errors. The sub-tasks are divided into two types: (1) *Type1*: Writing Verilog code for partial function/logic, and (2) *Type2*: Verifying and debugging the generated Verilog module. The code agent and debug agent are assigned to complete the *Type1* sub-task and *Type 2* sub-task, respectively. We first discuss the Verilog tools including a third-party simulator (i.e., iverilog (Williams and Baxter 2002)) and customized *AST-based waveform tracing tool*. Then, we introduce a code agent and a debug agent.

Verilog Tools The Verilog tools to assist agents for code implementation are listed below.

Syntax checker tool: We use iverilog to compile the generated Verilog code module and provide compiled messages as feedback for syntax checking.

Verilog simulator tool: We use iverilog to compile the generated Verilog code module and launch the Verilog simulation. If the generated Verilog code module contains syntax errors, the tool reports the lines where these errors occur. On the other hand, the tool also reports the simulation results, including the number of mismatches in output signals and the first mismatched time point. Additionally, the tool generates a VCD file format for waveform tracing.

AST-based waveform tracing tool (AST-WT): We developed a novel AST-based waveform tracing tool to assist agents in back-tracing the waveform of signals from mismatched output signals. Here, we extract the AST of generated Verilog module using Pyverilog library (Takamaeda-Yamazaki 2015). By inputting the mismatched output signals from the Verilog simulation tool and the desired back-tracing level, the tool starts from the mismatched signal and iteratively extracts the RVALUE signals until it reaches the specified back-tracing level in the AST, as the illustration shown in Figure 1(b). The back-tracing level parameter is determined dynamically by the AI agent through the Thought-Action-

Observation reasoning trace. The output includes the Verilog code reference, a tabular waveform of the mismatched signal, and the extracted RVALUE signals.

Code Agent For the code agent to write syntax-correct and consistent Verilog code, there are two LLMs: one acting as a Verilog Engineer and the other as a Verilog Verification Assistant, as shown in Figure 2(d). The Verilog Engineer writes the Verilog code according to the sub-task, while the Verilog Verification Assistant ensures that the written Verilog code is consistent with the sub-task requirements and free of syntax errors using the *syntax checker tool*. If there are syntax errors or inconsistencies between the written Verilog code and the sub-task description, the Verilog Verification Assistant will provide suggestions to the Verilog Engineer for fixing the issues. This process continues iteratively between the Verilog Engineer and the Verilog Verification Assistant until the generated Verilog code is free of syntax errors and consistent with the sub-task description.

Debug Agent The Debug Agent verifies the functionality and modifies the Verilog code to pass the functionality check from a provided testbench using collaborative Verilog verification tools as shown in Figure 2(d). Given the generated Verilog module from the previous task, the LLM-based Verilog Engineer performs reasoning and interacts with Verilog simulators, as well as the novel *AST-WT* through a Thought-Action-Observation process until the generated Verilog code passes the functionality check. Figure 4 shows an example of the Thought-Action-Observation process of the Verilog engineer fixing functionality issues through reasoning and interaction with *Verilog simulator tool* and *AST-WT*.

Experimental Results

Our work is implemented in Python and is built on top of the Autogen (Wu et al. 2023) multi-AI agent framework. We employ VerilogEval-Human v2 (Pinckney et al. 2024), which extends the 156 problems of VerilogEval-Human from (Liu et al. 2023a) to specification-to-RTL tasks, as our evaluation benchmark. We use the same planning, coding, and debugging prompts for these 156 problems. To check the functional correctness, the generated Verilog code is tested with the provided golden testbench. We measure Verilog functional correctness by running the VerilogCoder once for each problem in the benchmark.

Firstly, we demonstrate the Verilog functional correctness of prior works and the proposed VerilogCoder in the Main Results. Next, we conduct an ablation study on the impact of various types of planners and on the effect of using the proposed *AST-WT* for specification-to-RTL tasks.

Main Results

We demonstrate the pass-rates of the proposed method and prior works on the VerilogEval-Human v2 benchmark. We use OpenAI's GPT-4 Turbo (OpenAI 2024) and Llama3 (Meta 2024b) as the LLM models for the proposed VerilogCoder (Llama3) and VerilogCoder (GPT-4 Turbo), respectively, in the main experiment. The temperature and top_p parameters of the LLMs are set to 0.1 and 1.0, respectively. As we are the first to explore using an agentic method to generate functionally correct Verilog code, we compare the proposed VerilogCoder with recent LLMs using prompt engineering approaches. Table 1 shows the pass rates for RTL-Coder (Liu et al. 2023b), DeepSeek Coder (Guo et al. 2024), CodeGemma (CodeGemma Team, Google 2024), CodeLlama (Meta 2024a), Llama3 (Meta 2024b), Mistral Large (AI 2024), GPT-4 (OpenAI 2023), GPT-4 Turbo (OpenAI 2024), and the proposed VerilogCoder. For a fair comparison, we report the highest pass@1 score across 0-shot, 1-shot, and a sample size ranging from 1 to 20 on the Specification-to-RTL tasks from (Pinckney et al. 2024). For the VerilogEval-Human v2 benchmark, the proposed VerilogCoder (Llama3) successfully improves the Verilog coding ability of the open-source model and achieves 25.6% and 7.3% higher pass rates than Llama3 and GPT-4 Turbo with few-shot and in-context learning techniques (Pinckney et al. 2024), respectively. Moreover, the proposed VerilogCoder (GPT-4 Turbo) not only achieves a 94.2% pass rate but also outperforms the state-of-the-art recent LLMs GPT-4 and GPT-4 Turbo by 43.6% and 33.9%, respectively.

Method	Model Size	Model Type	Pass-Rate (%)
RTL-Coder	6.7B	Open	36.5
DeepSeek Coder	6.7B	Open	28.2
CodeGemma	7B	Open	23.1
DeepSeek Coder	33B	Open	37.2
CodeLlama	70B	Open	41.0
Llama 3	70B	Open	41.7
Mistral Large	Undisclosed	Closed	48.7
GPT-4	Undisclosed	Closed	50.6
GPT-4 Trubo	Undisclosed	Closed	60.3
VerilogCoder (Llama3)	70B	Open	**67.3**
VerilogCoder (GPT-4 Turbo)	Undisclosed	Closed	**94.2**

Table 1: Pass-rates of recent large language models (i.e., non-agentic method) and the proposed VerilogCoder. We run the VerilogCoder once for each problem in the benchmark. The pass-rates of VerilogCoder (agentic method) = #passed_case/#total_case. For the pass-rates of recent large language models, we report the best pass@1 score across 0-shot, 1-shot, and sample sizes ranging from 1 to 20 on the specification-to-RTL tasks from (Pinckney et al. 2024).

Here, the average number of group chat rounds for the high-level planner agent and the TCRG retrieval agent is 1.58 and 1.09, respectively. The code agent makes an average of 2.37 Verilog simulator tool calls and 1.37 *AST-WT* calls. The average token count of VerilogCoder is approximately 13× more than the GPT-4 Turbo baseline method.

Ablation Study

We conducted an ablation study to evaluate the impact of various types of planners, both with and without the proposed *AST-based waveform tracing tool*. We list two types of planners: (a) *Planner1*: A multi-LLM agent consisting of a planner and verilog engineer, and (b) *Planner2*: The proposed TCRG based task planner for task-oriented solving. In *Planner1*, given a module description or specification, the planner first decomposes the task into sub-tasks, and the Verilog engineer generates functionally correct Verilog code, including interactions with the provided Verilog verification

	Planner1	*Planner2*
without *AST-WT*	66.7% (baseline)	74.4% (7.7%)
with *AST-WT*	78.2% (11.5%)	94.2% (27.5%)

Table 2: Pass-rate (%) of Ablation study of *Planner1* without *AST-WT*, *Planner1* with *AST-WT*, *Planner2* without *AST-WT*, *Planner2* with *AST-WT*. AST-WT=AST-based waveform tracing tool. *Planner1* without *AST-WT* is the baseline, and *Planner2* with *AST-WT* is the proposed VerilogCoder.

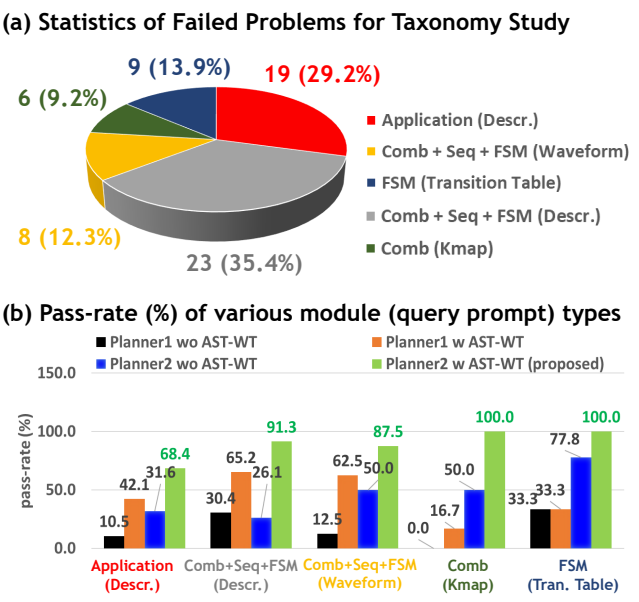

Figure 5: Taxonomy study results. (a) The statistics of extracted failed problems set and the number of problems in each module and query prompt type category. (b) Pass-rate (%) of each module and query prompt type categories.

tools. If syntax or functionality errors occur, the planner debugs and suggests alternative fixes for the Verilog engineer to correct the code. This iterative process between the planner and the Verilog engineer continues until the syntax and functionality are correct or the number of consecutive auto-replies in the group chat exceeds the maximum limit of 100.

Table 2 shows the pass-rates from the ablation study involving the combinations of *Planner1*, *Planner2*, and the proposed *AST-based waveform tracing tool* on the VerilogEval-Human v2 benchmark. With *Planner1*, the *AST-WT* achieves a 11.5% improvement in pass-rate. In contrast, *Planner2* without *AST-WT* improves by 7.7% compared to the baseline. Combining *Planner2* with *AST-WT*, as in the proposed VerilogCoder, significantly improves the pass-rate by 27.5% compared to the baseline.

To further investigate the reasons behind the significant improvement in the pass rate of the proposed VerilogCoder, we extract the union set of failed problems from the four combinations and categorize them based on the module and query prompt type of each failed problem for taxonomy study. Figure 5(a) shows the statistics of the number of each category and their description are listed below.

- Application (Descr.): The module is considered for an application (i.e., maze games, lemmings, timer, etc) with descriptions of its functionality in the query prompt.
- Comb+Seq+FSM (Descr.): The module is a block of combinational logic, sequential components, or finite state machine (FSM) with descriptions of its connections, and state transitions in the query prompt.
- Comb+Seq+FSM (Waveform): The module is a block of combinational logic, sequential components, or FSM with tabular waveform examples in the query prompt.
- Comb (Kmap): The module is a block of combinational logic with the Karnaugh map in the query prompt.
- FSM (Trans. Table): The module is a FSM block with the state transition table in the query prompt.

Figure 5(b) shows the pass-rate (%) of *Planner1* without *AST-WT*, *Planner1* with *AST-WT*, *Planner2* without *AST-WT*, and the proposed method. We observe that *Planner1* with *AST-WT* achieves 10.5%, 39.1%, and 12.5% higher pass-rates on the Application (Descr.), Comb+Seq+FSM (Descr.), and Comb+Seq+FSM (Waveform) categories than *Planner2* without *AST-WT*, respectively. The agent needs *AST-WT* to iteratively modify the generated Verilog code, as the indirect transformation from description and waveform to hardware description language may lead to confusion and misleading information. On the other hand, *Planner2* without *AST-WT* outperforms *Planner1* with *AST-WT* on the Comb (Kmap) and FSM (Trans. Table) tasks by 33.3% and 44.5%, respectively. This is because the proposed task planner can accurately capture the specified input-output mappings or state transitions in the plan without missing any information, ensuring that the code agent solves the subtasks step-by-step. Consequently, with the assistance of the proposed task planner and the *AST-based waveform tracing tool*, the proposed VerilogCoder can significantly improve the pass-rate across these types of tasks in the benchmark.

Conclusion and Future Work

Our proposed VerilogCoder has demonstrated the capability to autonomously write Verilog code and fix syntax and functional errors using the Verilog simulator and the proposed *AST-WT*. The ablation study reveals that the proposed novel TCRG-based task planner and task-oriented solving approach show a 7.7% improvement in pass-rate. Additionally, the proposed *AST-WT* achieves an 11.5% improvement in pass-rate. In summary, with the proposed TCRG based task planner and *AST-WT*, the proposed Verilogcoder achieves a 33.9% higher pass-rate compared to the state-of-the-art method.

We also believe that important directions for future Verilog agent-based research include: (1) training LLMs with high-quality Verilog code, (2) improving the generated Verilog code by considering PPA metrics, and (3) incorporating more efficient self-learning techniques and memory systems to enable the agent to accumulate experiences and continuously improve the quality of the generated Verilog code in terms of PPA metrics in the design flow.

References

AI, M. 2024. Au large. Section: news. https://mistral.ai/news/mistral-large/.

Alon, U.; Zilberstein, M.; Levy, O.; and Yahav, E. 2019. code2vec: Learning distributed representations of code. *Proceedings of the ACM on Programming Languages*, 3(POPL): 1–29.

Bairi, R.; Sonwane, A.; Kanade, A.; Iyer, A.; Parthasarathy, S.; Rajamani, S.; Ashok, B.; and Shet, S. 2024. Codeplan: Repository-level coding using llms and planning. *Proceedings of the ACM on Software Engineering*, 1(FSE): 675–698.

Bui, N. D.; Le, H.; Wang, Y.; Li, J.; Gotmare, A. D.; and Hoi, S. C. 2023. Codetf: One-stop transformer library for state-of-the-art code llm. *arXiv preprint arXiv:2306.00029*.

CodeGemma Team, Google. 2024. google/codegemma-7b · hugging face. https://huggingface.co/google/codegemma-7b.

crewAI Inc. 2024. crewAI: Cutting-edge framework for orchestrating role-playing, autonomous AI agents. By fostering collaborative intelligence, CrewAI empowers agents to work together seamlessly, tackling complex tasks. https://github.com/crewAIInc/crewAI.

Edge, D.; Trinh, H.; Cheng, N.; Bradley, J.; Chao, A.; Mody, A.; Truitt, S.; and Larson, J. 2024. From local to global: A graph rag approach to query-focused summarization. *arXiv preprint arXiv:2404.16130*.

Guo, D.; Zhu, Q.; Yang, D.; Xie, Z.; Dong, K.; Zhang, W.; Chen, G.; Bi, X.; Wu, Y.; Li, Y.; et al. 2024. DeepSeek-Coder: When the Large Language Model Meets Programming–The Rise of Code Intelligence. *arXiv preprint arXiv:2401.14196*.

Huang, D.; Bu, Q.; Zhang, J. M.; Luck, M.; and Cui, H. 2023. Agentcoder: Multi-agent-based code generation with iterative testing and optimisation. *arXiv preprint arXiv:2312.13010*.

Kommineni, V. K.; König-Ries, B.; and Samuel, S. 2024. From human experts to machines: An LLM supported approach to ontology and knowledge graph construction. *arXiv preprint arXiv:2403.08345*.

Liu, M.; Pinckney, N.; Khailany, B.; and Ren, H. 2023a. Verilogeval: Evaluating large language models for verilog code generation. In *2023 IEEE/ACM International Conference on Computer Aided Design (ICCAD)*, 1–8. IEEE.

Liu, S.; Fang, W.; Lu, Y.; Zhang, Q.; Zhang, H.; and Xie, Z. 2023b. Rtlcoder: Outperforming gpt-3.5 in design rtl generation with our open-source dataset and lightweight solution. *arXiv preprint arXiv:2312.08617*.

Mastropaolo, A.; Pascarella, L.; Guglielmi, E.; Ciniselli, M.; Scalabrino, S.; Oliveto, R.; and Bavota, G. 2023. On the robustness of code generation techniques: An empirical study on github copilot. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, 2149–2160. IEEE.

Meta. 2024a. meta-llama/CodeLlama-70b-instruct-hf · hugging face. https://huggingface.co/meta-llama/CodeLlama-70b-Instructhf.

Meta. 2024b. meta-llama/llama3. Original-date: 2024-03-15T17:57:00Z. https://github.com/meta-llama/llama3.

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

OpenAI. 2023. Gpt-4 technical report.

OpenAI. 2024. New models and developer products announced at DevDay. https://openai.com/index/new-models-and-developerproducts-announced-at-devday/.

OpenDevin Team. 2024. OpenDevin: An Open Platform for AI Software Developers as Generalist Agents. https://github.com/OpenDevin/OpenDevin.

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

Pinckney, N.; Batten, C.; Liu, M.; Ren, H.; and Khailany, B. 2024. Revisiting VerilogEval: Newer LLMs, In-Context Learning, and Specification-to-RTL Tasks. *arXiv preprint arXiv:2408.11053*.

Takamaeda-Yamazaki, S. 2015. Pyverilog: A Python-Based Hardware Design Processing Toolkit for Verilog HDL. In *Applied Reconfigurable Computing*, volume 9040 of *Lecture Notes in Computer Science*, 451–460. Springer International Publishing.

Thakur, S.; Ahmad, B.; Pearce, H.; Tan, B.; Dolan-Gavitt, B.; Karri, R.; and Garg, S. 2024. Verigen: A large language model for verilog code generation. *ACM Transactions on Design Automation of Electronic Systems*, 29(3): 1–31.

Tsai, Y.; Liu, M.; and Ren, H. 2023. Rtlfixer: Automatically fixing rtl syntax errors with large language models. *arXiv preprint arXiv:2311.16543*.

Wang, L.; Ma, C.; Feng, X.; Zhang, Z.; Yang, H.; Zhang, J.; Chen, Z.; Tang, J.; Chen, X.; Lin, Y.; et al. 2024. A survey on large language model based autonomous agents. *Frontiers of Computer Science*, 18(6): 186345.

Weng, L. 2023. LLM-powered Autonomous Agents. *lilianweng.github.io*.

Williams, S.; and Baxter, M. 2002. Icarus verilog: open-source verilog more than a year later. *Linux Journal*, 2002(99): 3.

Wu, Q.; Bansal, G.; Zhang, J.; Wu, Y.; Zhang, S.; Zhu, E.; Li, B.; Jiang, L.; Zhang, X.; and Wang, C. 2023. Autogen: Enabling next-gen llm applications via multi-agent conversation framework. *arXiv preprint arXiv:2308.08155*.

Yang, J.; Jimenez, C. E.; Wettig, A.; Lieret, K.; Yao, S.; Narasimhan, K.; and Press, O. 2024. Swe-agent: Agent-computer interfaces enable automated software engineering. *arXiv preprint arXiv:2405.15793*.

Yao, S.; Zhao, J.; Yu, D.; Du, N.; Shafran, I.; Narasimhan, K.; and Cao, Y. 2022. React: Synergizing reasoning and acting in language models. *arXiv preprint arXiv:2210.03629*.

Zhang, B.; and Soh, H. 2024. Extract, Define, Canonicalize: An LLM-based Framework for Knowledge Graph Construction. *arXiv preprint arXiv:2404.03868*.