MAGE: A Multi-Agent Engine for Automated RTL Code Generation

Yujie Zhao*, Hejia Zhang*, Hanxian Huang, Zhongming Yu, Jishen Zhao
University of California San Diego
La Jolla, CA 92093, USA
{yuz285, hez024, hah008, zhy025, jzhao}@ucsd.edu

Abstract—**The automatic generation of RTL code (e.g., Verilog) through natural language instructions has emerged as a promising direction with the advancement of large language models (LLMs). However, producing RTL code that is both syntactically and functionally correct remains a significant challenge. Existing single-LLM-agent approaches face substantial limitations because they must navigate between various programming languages and handle intricate generation, verification, and modification tasks. To address these challenges, this paper introduces MAGE, the first open-source[1] multi-agent AI system designed for robust and accurate Verilog RTL code generation. We propose a novel high-temperature RTL candidate sampling and debugging system that effectively explores the space of code candidates and significantly improves the quality of the candidates. Furthermore, we design a novel Verilog-state checkpoint checking mechanism that enables early detection of functional errors and delivers precise feedback for targeted fixes, significantly enhancing the functional correctness of the generated RTL code. MAGE achieves a 95.7% rate of syntactic and functional correctness code generation on VerilogEval-Human v2 benchmark, surpassing the state-of-the-art Claude-3.5-sonnet by 23.3%, demonstrating a robust and reliable approach for AI-driven RTL design workflows.**

I. Introduction

Digital hardware design usually requires engineers to define the architecture and functionality of hardware by writing code in hardware description languages (HDLs), such as Verilog and VHDL. As VLSI designs become more complex, hardware design workflows using HDLs are increasingly time-consuming and error-prone [1]–[3], often requiring multiple iterations to debug functional correctness. Although electronic design automation (EDA) tools have advanced to support this workflow [4]–[7], the demand for more efficient and automated hardware design solutions remains crucial.

Large Language Models (LLMs) [8], [9] have recently shown promising potential in assisting and improving hardware design [10]–[13]. Recent studies explored leveraging LLMs to improve the correctness of RTL code generation through model fine-tuning [10], [14], [15], training of domain-adaptive models [16], and single-agent methods [13], [17], [18] that augment LLMs by incorporating results and feedback from compilers or simulators to refine the generated code.

* Equal Contribution
[1]MAGE is open-sourced at https://github.com/stable-lab/MAGE-A-Multi-Agent-Engine-for-Automated-RTL-Code-Generation.

However, previous studies using a single LLM or a single agent face substantial limitations. First, these approaches fall short of the context-switch between tasks, programming languages, and knowledge domains. As a result, the quality of RTL design suffers: for instance, a single Claude-3.5-sonnet agent [9] has a functionality pass rate of only 75.0% even for simple design tasks [19]–[21]. Second, HDLs are specifically designed to describe the logic and architecture of digital hardware at a low level, focusing on the flow of data between registers, the timing of operations, and hardware characteristics such as propagation delays and signal dependencies. Directly adopting general-purpose LLM or multi-agent designs will produce low-accuracy RTL code that fails to meet timing constraints, resulting in wrong designs [10], [16], [19], [20].

To address these challenges, we propose **MAGE**, the first open-source **M**ulti-**AG**ent **E**ngine to achieve automatic high-quality RTL code generation. Different from the existing single-agent RTL code generation frameworks [13], [17], [18], MAGE enables specialized agents to handle distinct aspects in the RTL development pipeline. Inspired by real-world RTL design workflows, where specialists focus on distinct stages, our approach, MAGE, employs a specialized multi-agent architecture composed of four types of key agents: the RTL code generation agent, testbench generation agent, judge agent, and debug agent. Each type plays a specific role, working collaboratively within our tailored recursive framework to generate optimized and reliable RTL code.

MAGE consists of three key **design principles**. First, mimicking the iterative nature of human RTL design teams in addressing complex design challenges, we developed a high-efficiency collaboration workflow with a delicate design context communication protocol. Second, we propose a novel high-temperature RTL candidate sampling and debugging system, which leverages both high-temperature sampling and simulation-based scoring to identify promising candidates. Our key insight is that higher temperatures result in more diverse LLM outputs, while also increasing the likelihood of including the highest-quality candidates for optimization in the next stage. Finally, we propose a novel Verilog-state checkpointing and validation scheme, which substantially improves the functional correctness. Unlike conventional methods that provide feedback only on the final output mismatch [13], [17], [18], MAGE validates an RTL design by comparing state values with expected outputs at each clock edge. This enables early detection of functional errors and provides precise feedback for targeted fixes.

In summary, we make the following contributions:

- We design MAGE, the first open-source LLM-based multi-agent system for robust and accurate Verilog RTL code generation. By decomposing a complex hardware design into manageable sub-tasks, we design LLM agents to handle sub-tasks and design the system to enable effective agent communication and collaboration.
- We propose a novel high-temperature RTL candidate sampling system, notably enhancing generated code quality by exploring the benefits of high-randomness generation.
- We propose a novel Verilog-state checkpoint checking mechanism, which provides precise feedback for targeted fixes, significantly improving the RTL code quality.
- MAGE achieves a **95.7%** rate of syntactic and functional correctness code generation on the VerilogEval- Human v2 benchmark, surpassing all existing methods, representing a critical step toward automating and optimizing hardware design workflows, offering a more robust methodology for AI-driven RTL design.

II. Background and Motivation

A. Characteristics and challenges of LLM-based RTL design

Decomposing Complex Traditional RTL Design. Traditional digital hardware design flows necessitate hardware engineers to iteratively perform: (1) implement Verilog code to specify hardware architectures and behaviors, (2) customize test benches to rigorously verify the correctness of these hardware descriptions (3) iterate between Verilog simulations, signal waveform reasoning, and code refining until all output signals match expected behavior. This iterative loop among design, verification, and refinement makes the RTL design process not only challenging but also highly demanding in terms of both time and expertise. The complexity of traditional Verilog RTL design calls for decomposing the whole process into multiple manageable sub-stages and adopting different specialists (agents) for different sub-stages.

Related Work on LLM-based RTL Design. Recent works [10], [15], [16], [22], [23] train or fine-tune general code generation LLMs by incorporating RTL domain knowledge to enhance the correctness of RTL code generation. Single-agent methods [13], [17], [18] further integrate simulation results and introduce more stages in the code generation process, such as planning, verification, and code refining based on the feedback from simulation. However, this process involves generating both synthesizable and non-synthesizable code for RTL generation and verification, and context switches among different sub-tasks in multiple iterations, necessitating different domain knowledge and problem-solving abilities. Thus, adopting a single agent for all these tasks in the hardware design process leads to sub-optimal results, as they have to process such complex contexts in each interaction and maintain consistency through a long unified conversation history.

In contrast, multi-agent systems distribute tasks among agents with independent conversation histories, enabling specialized task handling and greater modularity. However, Aivril [24] only implements a basic two-agent division between code generation and review, still requiring a single agent to handle both synthesizable RTL and non-synthesizable verification code, thus failing to address the fundamental context-switching challenges. Verilogcoder [25] further constrains accessibility and system adaptability through its closed-source implementation and reliance on proprietary components (e.g., its Abstract Syntax Tree and Waveform Tracing Tool).

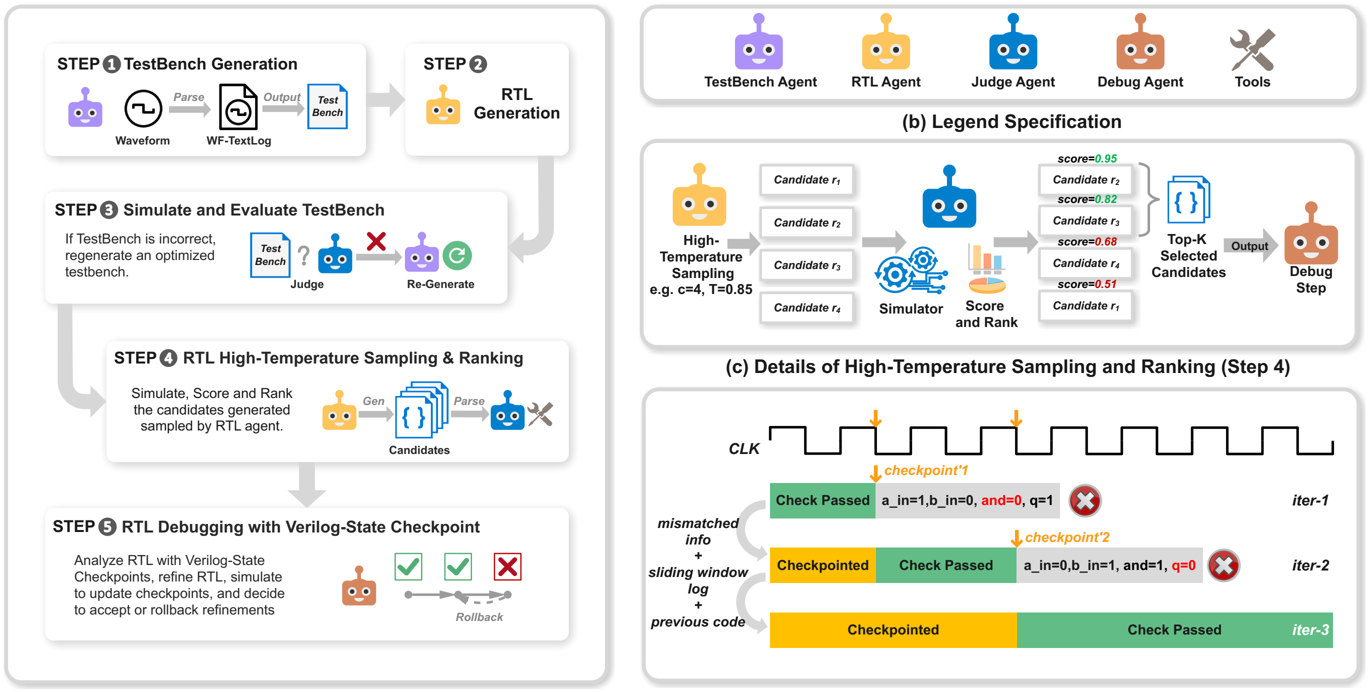
Temperature Sampling for LLM-based RTL Design. Given an input requirement, LLMs rely on a specific decoding strategy to generate the code auto-regressively. Temperature sampling method [26] uses a temperature coefficient T (usually $\in [0,1]$) to control the sampling randomness. Increasing the temperature avoids overly conservative results and promotes diversity in code generation, thus enhancing the chance of exploring the correct answers. However, this comes at the cost of introducing more noise and errors in the generation results. Recent studies [27], [28] show that high-temperature sampling can improve correctness in software code generation through multiple sampling iterations. However, higher temperatures tend to result in poorer performance for RTL design as explored in recent studies [10], [21]. We found a key reason for this is that single-agent mechanisms restrict the ability to design independent and efficient sampling and feedback, thereby hindering the optimization potential of high-temperature sampling for RTL code.

B. Opportunity on Effective Code Generation

To tackle the aforementioned limitations of both existing single-agent and multi-agent systems for RTL code generation, we identify potential opportunities to develop effective RTL code generation frameworks to improve correctness.

First, to further reduce the mutual influence between non-synthesizable testbenches and synthesizable RTL code generation, as well as to decrease the complexity of tasks assigned to each agent, we recognize the necessity of appropriately distributing tasks among multiple agents and organizing the workflow effectively. Specifically, previous works (e.g., [17], [24]) employed a single agent to generate both non-synthesizable testbenches and synthesizable RTL code simultaneously. However, [29] found that generating testbenches and code together within the same conversation may lead to a corresponding decrease in effectiveness. Moreover, test cases generated in this manner can become biased and influenced by the code, resulting in a loss of objectivity and diversity.

Second, based on our analysis in Sec. II-A, we thoroughly reviewed existing multi-agent systems [24], [25], where we found that all of them are closed-source and dependent on third-party proprietary tools that are not LLM-directly-adapted. This inevitably limits system extensibility and transparency. Therefore, we identify the need for an open-sourced multi-agent system.


STEP 1 TestBench Generation
Waveform
Parse
WF-TextLog
Output
Test Bench
STEP 2 RTL Generation
STEP 3 Simulate and Evaluate TestBench
If TestBench is incorrect, regenerate an optimized testbench.
Judge
Re-Generate
STEP 4 RTL High-Temperature Sampling & Ranking
Simulate, Score and Rank the candidates generated sampled by RTL agent.
Gen
Candidates
Parse
STEP 5 RTL Debugging with Verilog-State Checkpoint
Analyze RTL with Verilog-State Checkpoints, refine RTL, simulate to update checkpoints, and decide to accept or rollback refinements
Rollback
TestBench Agent
RTL Agent
Judge Agent
Debug Agent
Tools
High-Temperature Sampling e.g. c=4, T=0.85
Candidate r1
Candidate r2
Candidate r3
Candidate r4
Simulator
Score and Rank
score=0.95
Candidate r2
score=0.82
Candidate r3
score=0.68
Candidate r4
score=0.51
Candidate r1
Top-K Selected Candidates
Output
Debug Step
CLK
checkpoint'1
Check Passed
a_in=1,b_in=0, and=0, q=1
iter-1
mismatched info + sliding window log + previous code
checkpoint'2
Checkpointed
Check Passed
a_in=0,b_in=1, and=1, q=0
iter-2
Checkpointed
Check Passed
iter-3

(a) Overall architecture of MAGE (b) Legend Specification (c) Details of High-Temperature Sampling and Ranking (Step 4) (d) Details of RTL Debugging with State Checkpoint (Step 5)

Fig. 1: (a) The overview of MAGE, (b) the roles of four types of agents, (c) state check module, and (d) sampling and debugging module.

Finally, to address the limitation of high-temperature sampling for RTL code generation described in Sec. II-A, we summarize the key to efficiently sampling RTL code candidates process to mitigate the impact of randomness while leveraging the exploration benefits. Given that RTL code is synthesizable and a practical miscount scoring method is applicable, we prioritize candidate selection at an early stage by identifying top-scoring candidates. This approach allows us to reduce the exploration cost by filtering suboptimal results early.

C. *Opportunity on Effective Code Debugging*

Most existing LLM approaches for RTL code generation [13], [17], [18] rely on the direct application of original golden testbenches, which can only provide pass rates. This provides very limited feedback to LLMs in fine-tuning RTL code as pass rates alone lack detailed insights into crucial aspects such as timing analysis, signal interactions, variable values, and mismatch information, all of which are essential for ensuring high-quality RTL design. VerilogCoder [25] applies closed-source Abstract Syntax Tree (AST) analysis, which significantly restricts flexibility and transparency. Furthermore, the introduction of tools that output in graphical form, which LLM can not directly apply, increases the complexity of tasks for LLMs and reduces their effectiveness. Therefore, we identify the requirement for an optimized testbench, which will output a log resembling a simulated waveform in text form, which can be directly adaptable by LLMs. This shifts the reliance on closed-source tools to a fully open-source LLM-based textual protocol, alongside significant improvements in LLM RTL code debug effectiveness, improving both the extension of the framework and the quality of the analysis.

III. MAGE Design

We present MAGE, a multi-agent engine designed specifically for RTL. This system integrates an RTL-specific context communication protocol with high-efficiency, LLM-adapted tools for fine-tuning RTL, all within a productive and orchestrated process.

A. *Multi-Agent System Overview*

Figure 1 depicts a comprehensive overview of the workflow implemented in MAGE. Inspired by the collaborative division of work approaches of human RTL design teams, our framework incorporates four types of agents, each playing a specific and concrete role in the automation process. Figure 1 (b) specifically delineates the responsibilities and tasks assigned to each agent, ensuring a clear understanding of the workflow. (1) A Testbench Generation Agent is responsible for creating optimized test benches in a textual-waveform-output format based on the natural language specifications and any available golden test benches. (2) A RTL Generation Agents convert these specifications along with the optimized test bench into Verilog code, incorporating syntax checking to ensure code validity. (3) A Judge Agent then takes over by simulating and evaluating the generated RTL code against the optimized test bench. It scores the RTL code candidates and decides whether any require debugging or if the test bench needs to be regenerated. (4) A Debug Agent performs iterative refinements on any code that fails initial tests, using textual waveform-like simulation outputs as feedback for improvements. Our method mimics the division of labor in human teams and enhances the efficiency and accuracy of automated RTL design. It should be noted that, whether in RTL code generation or debugging, the agent will perform at most $s = 5$ iterations to automatically fix syntax errors.

Based on the distinct and specific roles of multiple agents, we have designed an efficient workflow to ensure that each agent's unique contributions are seamlessly integrated, facilitating a coherent progression throughout the workflow. Figure 1 (a) demonstrates the entire process is systematically divided into five main steps as depicted as follows:

Step ❶ – Generate initial textual-waveform-output testbenches. To reduce the limitations of pass-rate-output golden testbenches, which is analyzed in Sec. II-C, we directly utilize natural language specifications to generate optimized testbenches that can output State Checkpoints, which will be used in Step ❺ (see Figure 1 (d) and Sec. III-C). Furthermore, since natural language specifications may contain ambiguities, we combine the input with the golden testbench if available.

Step ❷ – Based on the natural language specifications and the optimized testbench, we generate an initiate RTL code.

Step ❸ – If the initial RTL code cannot pass the optimized testbench, the judge agent evaluates the testbench and regenerates it if deemed incorrect.

Step ❹ – If the RTL code is deemed correct, we will employ an **High-Temperature RTL Sampling and Scoring Process** (see Figure 1 (c) and detailed in Sec. III-B) to generate and ranking RTL code Candidates.

Step ❺ – If the RTL code candidates still have function errors, we will employ a **RTL Debugging with State Checkpoint Mechanism** (see Figure 1 (d) and Sec. III-C) to make effective debug trials and debug the selected RTL code candidates.

B. High-Temperature RTL Sampling and Scoring

As depicted in Sec. II-B, high-temperature sampling has a better chance of exploring the correct code. However, in prior studies on Temperature Sampling for RTL code generation like [27], [28], it is commonly acknowledged that high-temperature sampling introduces significant randomness, often resulting in reduced correctness. To address this issue, we first explore the potential opportunity of using multiple RTL code generation candidates in high-temperature sampling to reduce simulated mismatch counts in one iteration. Fig. 2 shows the distribution of normalized mismatch counts for the best candidates at two different temperatures. We observe that the best candidate with high temperature has a lower mismatch count for most problems. This suggests that, despite increased randomness, high-temperature sampling yields better performance when the sampling is sufficiently extensive. The improvement in pass rates at each stage, as shown in Fig. 2, indicates the potential benefits of performing an effective sampling policy at high temperatures.

Based on the above observation, we propose a High-Temperature Sampling and Ranking Process as follows:

1. We sample c RTL code candidates for issue i:

$$\mathcal{R}_i = \{r_i^1, r_i^2, \ldots, r_i^c\}, r_i \sim P_T\left(r \mid p_{sys}, SP_i, TB_i\right) \quad (1)$$

Here, T is the temperature, p_{sys} denotes the system prompt of the RTL code generation, SP_i is the natural language specification of the issue i, and TB_i is the corresponding testbench. The term $P_T\left(r \mid p_{sys}, SP_i, TB_i\right)$ represents the probability distribution of the generated RTL code given the temperature, system prompt, specification, and testbench.

2. We select the Top-K candidates from $\mathcal{R}_i$ based on their scores. Specifically, each candidate in $\mathcal{R}_i$ is evaluated to obtain a normalized mismatch-based score as follows:

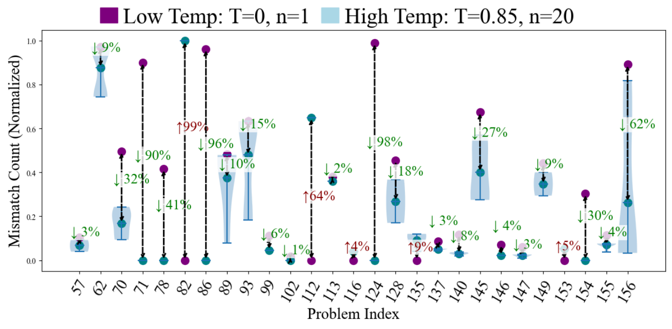
$$s(r) = 1 - \frac{m(r)}{tc(r)} \quad (2)$$

where $m(r)$ is the mismatch count and $tc(r)$ is the total number of checks of the generated RTL code r. We then select the K candidates with the highest scores, denoted as follows:

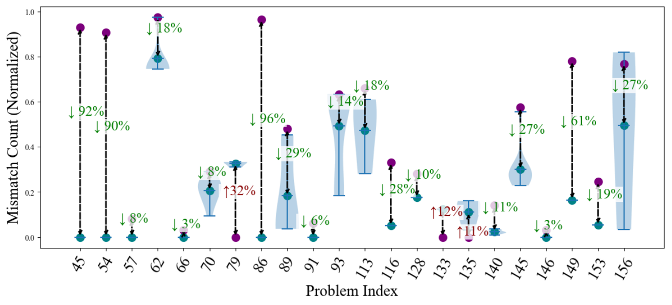
$$\mathcal{R}^{\star}_{i,0} = \underset{A \subseteq \mathcal{R}_i, |A| = K}{\arg\max} \sum_{r \in A} s(r) \quad (3)$$

3. We employ a debug agent to perform debugging trials, generating a new candidate $r^{\star}_{trial} = D(r^{\star})$ for each selected candidate $r^{\star}$, where $D(r)$ denotes the debug trial for RTL code r. The selected candidate set at iteration m is then updated as:

$$R^{\star}_{i,m} = \{ \underset{r \in \{D(r^{\star}), r^{\star}\}}{\arg\max} \; s(r) \mid r^{\star} \in \mathcal{R}^{\star}_{i,m-1} \} \quad (4)$$



(a) VerilogEval-v1



(b) VerilogEval-v2

Fig. 2: Normalized mismatch count of generated testbenches at different stages under varying temperature configurations (Low temperature: $T = 0$, $n = 1$; High temperature: $T = 0.85$, $n = 20$), using the Claude 3.5 Sonnet model (dated 2024-10-22) across two benchmarks: VerilogEval-v1-Human [20] and VerilogEval-v2 [21]. Problems that directly passed before Step ❹ and those with zero mean mismatches in both configurations are not shown in the figure. The blue violin plot shows that candidates (blue dots) generated with high-temperature sampling typically have lower mean mismatch counts across most problems compared to those generated with low temperatures (purple dots).

And this process repeats until $\max_{r \in R^{\star}_{i,m}} s(r) = 1$ or the iteration limit is reached.

C. RTL Debugging with State Checkpoint Mechanism

Our objective is to design a mechanism that is entirely LLM-based, independent of third-party closed-source tools, and effective in improving RTL debugging efficiency. To achieve this, we propose the RTL Debugging with State Checkpoint Mechanism, which follows these steps:
1. Utilizing the optimized testbench generated in Step ❶ and ❷, we extract the input vector $\mathbf{I}$, the DUT, and the output $\mathbf{O}$, and subsequently identify the earliest mismatch point as:

$$t_m = \min\{t \mid \mathbf{O}_{DUT}(t) \neq \mathbf{O}_{exp}(t), t \geq 0\} \tag{5}$$

where $\mathbf{O}_{\mathrm{DUT}}(t)$ is the output vector from the DUT at time t, and $\mathbf{O}_{\mathrm{exp}}(t)$ is the expected output vector at time t.
2. We generate the State Checkpoint and collect the textual waveform window as:

$$W = \Big\{ \big(\mathbf{I}(t'), \mathbf{O}_{DUT}(t'), \mathbf{O}_{exp}(t')\big) \mid t' \in \big[\max(t_m - L_W, 0), t_m\big] \Big\} \tag{6}$$

where W represents the textual waveform window, L_W is the window length parameter, and $\mathbf{I}(t')$ is the input vector at clock edge t'.
3. The debugging agent then takes the textual waveform window W and the original testbench as inputs, generates a new trial of debugged RTL code, and performs replacement actions to correct the identified faults in the RTL. The simulation is subsequently rerun to verify if the mismatch is resolved in the updated State Checkpoint.

IV. Experiments

A. Experimental Setup

Benchmarks. Two widely used benchmarking datasets: VerilogEval-v1-Human [20] and VerilogEval-v2 [25]. **Model.** Claude 3.5 Sonnet 2024-10-22 [9]. **Baselines.** i. Vanilla models: Generating RTL code in a single pass using language models, including both general-purpose LLMs (e.g. GPT-4o [30], Claude 3.5 Sonnet [9]) and RTL-specified models (e.g. ITERTL [22], CodeV [23]). ii. LLM Agent systems: Agent-based systems designed to enhance LLM performance for RTL code generation, including open-source solutions (OriGen [13]) and closed-source ones (e.g. VeriAssist [17], AutoVCoder [18], VerilogCoder [25], AIVRIL [24]). **Configurations.** Our implementation of MAGE integrates the open-source Verilog compiler and simulator Icarus Verilog [31] with an LLM-agnostic API interface offered by the open-source framework LlamaIndex [32]. Based on the superior performance [24], [33], [34] of Claude 3.5 Sonnet [9] in multiple coding tasks, we choose to run experiments with that language model. In accordance with the VerilogEval V1 [20] and V2 [21] benchmark, we conducted experiments to measure Pass@1 under 2 settings: *Low Temperature* (T=0, top_p=0.01, n=1) and *High Temperature* (T=0.85, top_p=0.95, n=20). Here, temperature T is described in Sec. II-A. Top_p limits the output pool to a cumulative probability threshold. n represents the number of evaluation runs. **Metrics.** Following prior works [17], [18], [21], the Pass@1 metric is computed as:

$$pass@k = \mathbb{E}_{\text{Problems}} \left[1 - \frac{\binom{n-c_p}{k}}{\binom{n}{k}} \right] \tag{7}$$

where $k = 1$, and c_p is the number of passing runs. This Pass@1 metric, which accounts for multiple runs, reflects the expected percentage of problems that the system solves correctly when executed once for each problem.

TABLE I: Pass rates of different temperature configurations in MAGE.

Config	VerilogEval-Human Pass@1	VerilogEval-V2 Pass@1
High Temp	**94.8**	**95.7**
Low Temp	89.1	93.6

As shown in Table I, the High Temperature setting achieves higher Pass@1, so it is adopted for subsequent experiments.

B. Key Results

Table II shows the comparison of MAGE and baselines. For a fair comparison, we select the highest pass rate among their experiment configuration. MAGE achieves the best performance on both benchmarks, delivering consistent improvements over specialized (e.g., VerilogCoder) and general-purpose systems(e.g., GPT-4, CodeQwen). Specifically, MAGE obtains a Pass@1 score of 94.8% on VerilogEval-Human and 95.7% on VerilogEval-V2, surpassing all baselines. These results underline the effectiveness of MAGE in advancing the capabilities of LLM-driven coding systems.

MAGE not only outperforms closed-source systems but also demonstrates substantial improvement over accessible solutions, including vanilla models and open-source coding systems. For instance, compared to the highest-performing vanilla LLMs (e.g., Claude 3.5 Sonnet), MAGE achieves a relative improvement of +19.8% on VerilogEval-Human and +23.3% on VerilogEval-V2. Similarly, against open-source systems like OriGen (54.4% Pass@1 on VerilogEval-Human), MAGE significantly improves the functional correctness, showcasing its superior performance in scenarios where accessible and reproducible solutions are prioritized.

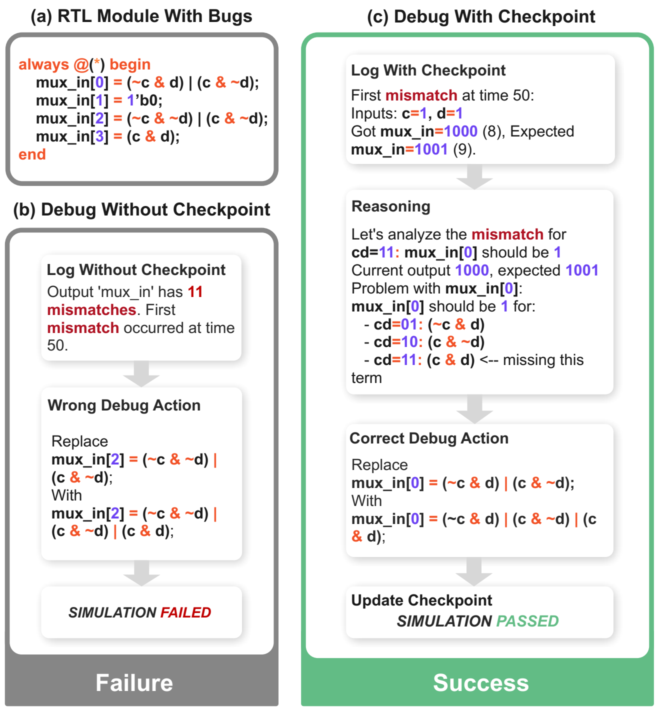
C. Ablation Study

Multi-Agent System. We conducted an ablation study to assess the effectiveness of task distribution among multiple agents. The study compared three configurations: (a) Vanilla LLM, which involves one-pass RTL code generation using a single LLM; (b) Single-Agent, where different agents in the MAGE system were merged into a single agent by sharing a common generation history; and (c) Multi-Agent, the proposed system that assigns tasks to specialized agents based on their roles. As shown in Table III, the multi-agent configuration achieves the highest pass rate (93.6%), outperforming both the vanilla (72.4%) and single-agent (83.9%) setups. These

TABLE II: Pass rates of recent LLMs and Coding Systems. The highest Pass@1 among different temperature settings is reported.

System	System Type	LLM Model	VerilogEval-Human Pass@1	VerilogEval-V2 Pass@1
Generic LLM	N/A	GPT-4o	51.3	N/A
	N/A	Claude 3.5 Sonnet*	60.3	N/A
	N/A	Claude 3.5 Sonnet 2024-10-22	75.0	72.4
RTL-specified LLM	N/A	ITERTL [22]	42.9	N/A
	N/A	CodeV [23]	53.2	N/A
OriGen [13]	**Open Source**	DeepSeek-Coder-7B + LoRA	54.4	N/A
VeriAssist [17]	Closed Source	GPT-4	50.5	N/A
AutoVCoder [18]	Closed Source	CodeQwen1.5-7B	48.5	N/A
VerilogCoder [25]	Closed Source	GPT-4 Turbo	N/A	94.2
AIVRIL [24]	Closed Source	Claude 3.5 Sonnet*	64.7	N/A
MAGE (ours)	**Open Source**	Claude 3.5 Sonnet 2024-10-22	**94.8**	**95.7**
		Improvement(Δ)[†]	+19.8	+23.3

* Claude 3.5 Sonnet has two versions: 2024-06-20 and 2024-10-22. The cited paper [24] does not specify which version was used.
[†] The improvement is directly compared to the performance of the same model without employing the MAGE system.



```
always @(*) begin
    mux_in[0] = (~c & d) | (c & ~d);
    mux_in[1] = 1'b0;
    mux_in[2] = (~c & ~d) | (c & ~d);
    mux_in[3] = (c & d);
end
```

Log Without Checkpoint
Output 'mux_in' has 11 mismatches. First mismatch occurred at time 50.

Wrong Debug Action
Replace
mux_in[2] = (~c & ~d) | (c & ~d);
With
mux_in[2] = (~c & ~d) | (c & ~d) | (c & d);

SIMULATION FAILED

Log With Checkpoint
First mismatch at time 50:
Inputs: c=1, d=1
Got mux_in=1000 (8), Expected mux_in=1001 (9).

Reasoning
Let's analyze the mismatch for cd=11: mux_in[0] should be 1
Current output 1000, expected 1001
Problem with mux_in[0]:
mux_in[0] should be 1 for:
- cd=01: (~c & d)
- cd=10: (c & ~d)
- cd=11: (c & d) <-- missing this term

Correct Debug Action
Replace
mux_in[0] = (~c & d) | (c & ~d);
With
mux_in[0] = (~c & d) | (c & ~d) | (c & d);

Update Checkpoint
SIMULATION PASSED

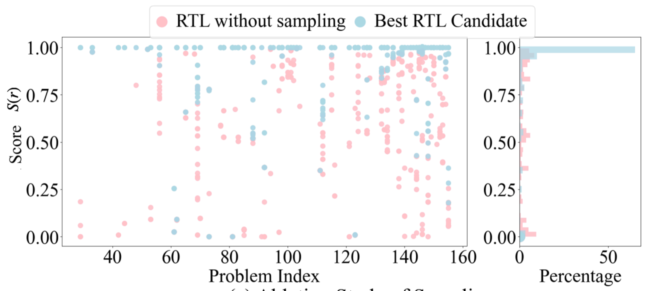
Fig. 3: The Case Study of RTL Code State Checkpoint on Prob093-ece241-2014-q3.

results indicate that effective task partitioning can significantly enhance the performance of LLM-based systems, particularly for complex tasks like RTL code generation, which require handling both synthesizable RTL code and non-synthesizable testbenches.

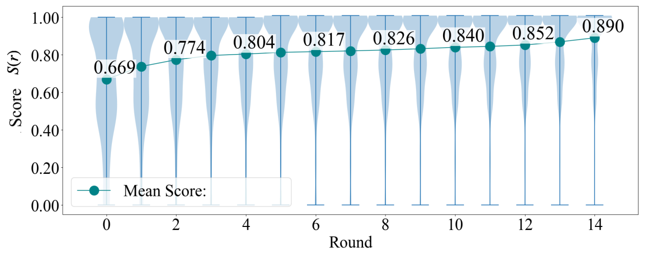
RTL Code State Checkpoint Mechanism. We also conducted case studies to evaluate the effectiveness of the proposed RTL Code State Checkpoint mechanism. For example, Fig. 3 illustrates a debugging case study with and without state checkpoints. Without checkpoints, when only the log is provided to the LLM Debug Agent, the agent can only approximate

TABLE III: Multi-Agent task distribution ablation study: Pass Rates with Claude 3.5 Sonnet 2024-10-22, Low-Temperature Setting.

Config	Type	VerilogEval-V2 Pass@1
Vanila LLM	Pass%	72.4
Single-Agent	Pass%	83.9
	Improvement(Δ)	**+11.5**
Multi-Agent	Pass%	93.6
	Improvement(Δ)	**+21.2**



(a) Ablation Study of Sampling



(b) Ablation Study of Debugging

Fig. 4: Score $S(r)$ improvement of RTL by sampling and debugging. (a) Score distribution: generated RTL without sampling versus sampled and selected best RTL candidate; (b) Score distribution and the mean score of generated RTL in each debug round. Data of problems fixed before entering the debug stage are not included.

the problematic location in the buggy RTL code, making it unlikely to identify and fix the issue. In contrast, with state checkpoints included in the log, the LLM Debug Agent can reason more effectively, pinpointing mismatches in the output and identifying missing logic terms. This leads to a more accurate and efficient bug-fixing process.

Sampling and Debugging Mechanisms. Our experimental data demonstrates the effectiveness of the proposed sampling and debugging mechanisms in enhancing the quality of generated RTL. Fig. 4(a) illustrates that the RTL generated with the sampling strategy consistently outperforms the RTL without sampling across different problems. The score distribution indicates that, without sampling, the RTL scores are nearly uniformly spread across the range $[0, 1]$. However, after applying the sampling method, the scores are concentrated near 1, reflecting a significant quality improvement. Fig. 4 (b) depicts the improvement in the mean score across multiple rounds of debugging, starting from an initial score of 0.669 and reaching 0.890 after sufficient refinement. This consistent increase in score demonstrates the cumulative benefit of iterative debugging, leading to more optimal RTL solutions.

V. Conclusion

In this paper, we propose MAGE, the first open-source LLM-based multi-agent system designed for automated and accurate Verilog RTL code generation. Integrated with a novel High-Temperature RTL Sampling and Scoring process, MAGE effectively explores more potentially correct candidates, leading to higher pass rates than prior studies. Augmented with RTL Debugging with State Checkpoint Mechanism, MAGE further optimizes the code with more precise feedback. Our system represents a critical step toward automating and optimizing hardware design workflows, offering a more robust methodology for AI-driven RTL design.

References

[1] G. Dessouky, D. Gens, P. Haney, G. Persyn, A. Kanuparthi, H. Khattri, J. M. Fung, A.-R. Sadeghi, and J. Rajendran, "HardFails: Insights into software-exploitable hardware bugs," in *28th USENIX Security Symposium (USENIX Security 19)*, 2019, pp. 213–230.

[2] B. Ahmad, S. Thakur, B. Tan, R. Karri, and H. Pearce, "On hardware security bug code fixes by prompting large language models," *IEEE Transactions on Information Forensics and Security*, 2024.

[3] K. Laeufer, B. Fajardo, A. Ahuja, V. Iyer, B. Nikolić, and K. Sen, "RTL-repair: Fast symbolic repair of hardware design code," in *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, ser. ASPLOS '24. New York, NY, USA: Association for Computing Machinery, 2024, p. 867–881. [Online]. Available: https://doi.org/10.1145/3620666.3651346

[4] L. Lavagno, L. Scheffer, and G. Martin, *EDA for IC implementation, circuit design, and process technology*. CRC press, 2018.

[5] L. Scheffer, L. Lavagno, and G. Martin, *EDA for IC system design, verification, and testing*. CRC press, 2018.

[6] K. Wakabayashi and T. Okamoto, "C-based soc design flow and eda tools: An asic and system vendor perspective," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 19, no. 12, pp. 1507–1522, 2000.

[7] R. Nane, V.-M. Sima, C. Pilato, J. Choi, B. Fort, A. Canis, Y. T. Chen, H. Hsiao, S. Brown, F. Ferrandi *et al.*, "A survey and evaluation of fpga high-level synthesis tools," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 35, no. 10, pp. 1591–1604, 2015.

[8] J. Achiam, S. Adler, S. Agarwal, L. Ahmad, I. Akkaya, F. L. Aleman, D. Almeida, J. Altenschmidt, S. Altman, S. Anadkat *et al.*, "Gpt-4 technical report," *arXiv preprint arXiv:2303.08774*, 2023.

[9] Anthropic, "The claude 3 model family: Opus, sonnet, haiku," 2024. [Online]. Available: https://www-cdn.anthropic.com/de8ba9b01c9ab7cbabf5c33b80b7bbc618857627/Model_Card_Claude_3.pdf

[10] S. Thakur, B. Ahmad, Z. Fan, H. Pearce, B. Tan, R. Karri, B. Dolan-Gavitt, and S. Garg, "Benchmarking large language models for automated verilog rtl code generation," in *2023 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2023, pp. 1–6.

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

[12] Y. Tsai, M. Liu, and H. Ren, "Rtlfixer: Automatically fixing rtl syntax errors with large language model," in *Proceedings of the 61st ACM/IEEE Design Automation Conference*, 2024, pp. 1–6.

[13] F. Cui, C. Yin, K. Zhou, Y. Xiao, G. Sun, Q. Xu, Q. Guo, D. Song, D. Lin, X. Zhang *et al.*, "Origen: Enhancing RTL code generation with code-to-code augmentation and self-reflection," *arXiv preprint arXiv:2407.16237*, 2024.

[14] H. Pearce, B. Tan, and R. Karri, "Dave: Deriving automatically verilog from english," in *Proceedings of the 2020 ACM/IEEE Workshop on Machine Learning for CAD*, 2020, pp. 27–32.

[15] S. Liu, W. Fang, Y. Lu, J. Wang, Q. Zhang, H. Zhang, and Z. Xie, "Rtlcoder: Fully open-source and efficient llm-assisted rtl code generation technique," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2024.

[16] M. Liu, T.-D. Ene, R. Kirby, C. Cheng, N. Pinckney, R. Liang, J. Alben, H. Anand, S. Banerjee, I. Bayraktaroglu *et al.*, "ChipNemo: Domain-adapted LLMs for chip design," *arXiv preprint arXiv:2311.00176*, 2023.

[17] H. Huang, Z. Lin, Z. Wang, X. Chen, K. Ding, and J. Zhao, "Towards LLM-powered Verilog RTL assistant: Self-verification and self-correction," *arXiv preprint arXiv:2406.00115*, 2024.

[18] M. Gao, J. Zhao, Z. Lin, W. Ding, X. Hou, Y. Feng, C. Li, and M. Guo, "Autovcoder: A systematic framework for automated Verilog code generation using llms," *arXiv preprint arXiv:2407.18333*, 2024.

[19] Y. Lu, S. Liu, Q. Zhang, and Z. Xie, "RTLLM: An open-source benchmark for design rtl generation with large language model," in *2024 29th Asia and South Pacific Design Automation Conference (ASP-DAC)*. IEEE, 2024, pp. 722–727.

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

[21] N. Pinckney, C. Batten, M. Liu, H. Ren, and B. Khailany, "Revisiting verilogeval: Newer LLMs, in-context learning, and specification-to-RTL tasks," *arXiv preprint arXiv:2408.11053*, 2024.

[22] P. Wu, N. Guo, X. Xiao, W. Li, X. Ye, and D. Fan, "Itertl: An iterative framework for fine-tuning llms for rtl code generation," *arXiv preprint arXiv:2407.12022*, 2024.

[23] Y. Zhao, D. Huang, C. Li, P. Jin, Z. Nan, T. Ma, L. Qi, Y. Pan, Z. Zhang, R. Zhang *et al.*, "Codev: Empowering LLMs for verilog generation through multi-level summarization," *arXiv preprint arXiv:2407.10424*, 2024.

[24] H. Sami, P.-E. Gaillardon, V. Tenace *et al.*, "Aivril: Ai-driven rtl generation with verification in-the-loop," *arXiv preprint arXiv:2409.11411*, 2024.

[25] C.-T. Ho, H. Ren, and B. Khailany, "Verilogcoder: Autonomous Verilog coding agents with graph-based planning and abstract syntax tree (AST)-based waveform tracing tool," *arXiv preprint arXiv:2408.08927*, 2024.

[26] D. H. Ackley, G. E. Hinton, and T. J. Sejnowski, "A learning algorithm for boltzmann machines," *Cognitive science*, vol. 9, no. 1, pp. 147–169, 1985.

[27] Q. Zheng, X. Xia, X. Zou, Y. Dong, S. Wang, Y. Xue, Z. Wang, L. Shen, A. Wang, Y. Li *et al.*, "Codegeex: A pre-trained model for code generation with multilingual evaluations on humaneval-x," *arXiv preprint arXiv:2303.17568*, 2023.

[28] Y. Zhu, J. Li, G. Li, Y. Zhao, Z. Jin, and H. Mei, "Hot or cold? adaptive temperature sampling for code generation with large language models," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 38, no. 1, 2024, pp. 437–445.

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

[30] OpenAI, "Hello GPT-4o," 2024, [Online; accessed 15-November-2024]. [Online]. Available: https://openai.com/index/hello-gpt-4o/

[31] S. Williams, "The icarus verilog compilation system," 2008. [Online]. Available: https://github.com/steveicarus/iverilog

[32] J. Liu, "LlamaIndex," 11 2022. [Online]. Available: https://github.com/jerryjliu/llama_index

[33] Anthropic, "Raising the bar on SWE-bench Verified with Claude 3.5 Sonnet," 2024, [Online; accessed 15-November-2024]. [Online]. Available: https://www.anthropic.com/research/swe-bench-sonnet

[34] S. Swaroopa, R. Mukherjee, A. Debnath, and R. S. Chakraborty, "Evaluating large language models for automatic register transfer logic generation via high-level synthesis," *arXiv preprint arXiv:2408.02793*, 2024.