MARS: A Multi-Agent RTL Synthesis Framework

Likith Anaparty[1,2*], Ananthakrishnan Thulasiraman[1*], Minghao Shao[3,4], Vivek Chaturvedi[1], Muhammad Shafique[4]
[1]Indian Institute of Technology (Palakkad), India [2]Jurin AI Inc., Japan
[3]NYU Tandon School of Engineering, USA [4]NYU Abu Dhabi, UAE
likith.79an@gmail.com, 122301003@smail.iitpkd.ac.in, vivek@iitpkd.ac.in
{shao.minghao, muhammad.shafique}@nyu.edu

***Abstract*—Automating Register Transfer Level (RTL) designs from natural language specifications remains a long-standing challenge in electronic design automation (EDA), where correctness-by-construction remains critical yet elusive. Despite the rapid progress in Large Language Models (LLMs), existing approaches focus more on synthesizing single-module RTL and underperform in generating functionally correct multi-module RTL designs. A key limitation lies in treating specifications as flat and monolithic units overlooking the modular structure, hierarchical dependencies and iterative refinement process fundamental to real-world RTL development. In this work, we introduce MARS, a novel multi-agent LLM framework that breaks down a hardware specification into sub-modules enabling specialized agents to focus on ambiguity resolution, complexity classification, design decomposition, per-module RTL generation, and integration. Agents collaborate through a coordinated pipeline producing modular testbench-aligned and verifiable RTL. Experimental results on RTL benchmarks demonstrate that MARS outperforms existing state-of-the-art baseline framework by 5.4% in generating functionally correct RTL designs.**

***Index Terms*—RTL synthesis, large language models, multi-agent systems, hardware design automation, Verilog generation**

I. Introduction

Digital hardware systems form the computational backbone of modern technologies. Designing such systems involves the specification and implementation of functional behavior at the Register Transfer Level (RTL), typically using hardware description languages (HDLs) such as SystemVerilog or VHDL. RTL remains a critical abstraction in electronic design automation (EDA), offering a balance between software-like expressiveness and synthesizability into gate-level netlists suitable for fabrication. This process requires deep domain knowledge, careful interface planning, and iterative refinement, especially for systems comprising multiple interacting modules.

Multiple research directions have emerged to streamline the RTL design process. For instance, High-Level Synthesis (HLS) tools allow designers to specify functionality using languages such as C or Python. While useful in early design phases, HLS often abstracts away critical low-level control over timing, resource utilization, and interface behavior that are essential in industrial workflows [1]. Recent advancements in large language models (LLMs) have introduced new opportunities in RTL design automation [2], [3]. These models exhibit strong capabilities in generating structured code and have shown encouraging results in synthesizing SystemVerilog from natural-language prompts, particularly for small-scale, self-contained modules. However, existing approaches tend to treat specifications as flat and monolithic, overlooking the inherently modular and hierarchical nature intrinsic to real-world hardware design. As a result, they struggle with functional correctness in tasks that require interface coordination, control logic synchronization, hierarchical decomposition [4], etc.

The current landscape of LLM-based RTL synthesis reveals two complementary axes for advancement: model-centric improvements and system-centric framework design. The model-centric axis focuses on enhancing the intrinsic capabilities of LLMs through domain-specific fine-tuning on HDL datasets, increased model parameter count, and the use of structured prompting strategies to improve reasoning and code generation [5]–[8]. Prior work has demonstrated that such approaches can substantially improve syntactic validity and coverage of common RTL constructs, and can yield modest gains in functional correctness for simple and moderately complex design tasks [5], [9]. However, this direction remains constrained by fundamental limitations. Publicly available HDL datasets for different hardware design purposes are limited in both scale and diversity compared to software codebases, restricting the effectiveness of large-scale fine-tuning and generalization [10]–[12]. Moreover, many real-world RTL design tasks require reasoning over long-range dependencies spanning multiple modules, shared signals, and control states. Accurately modeling such dependencies remains a challenge for LLMs, particularly in the absence of such sufficiently rich and representative training data [4].

In contrast, the system-centric axis emphasizes structured frameworks that guide, constrain, and validate LLM outputs. In this direction, prior research has introduced frameworks such as feedback-driven pipelines that iteratively refine RTL using compiler and simulation traces [13]–[15]. Researchers have also explored autonomous orchestration strategies that synthesize modules sequentially while propagating interface context and design state across stages [4]. Recently, MAGE [16] proposed a fully autonomous multi-agent framework that separates RTL generation, testbench generation, and simulation-based evaluation into distinct roles, enabling iterative refinement via simulation and state-checkpoint feedback. While

*Primary Authors

these approaches improve robustness compared to single-pass generation, they remain limited in their ability to capture the structural realities of practical RTL development. In particular, they don't reliably capture long-range dependencies across multiple modules, nor do they consistently enforce coherence over shared interfaces, signals, and control assumptions. As a result, designs that appear locally correct at the module level frequently fail during integration, exposing interface mismatches, protocol violations, or cross-module logic inconsistencies. These failures underscore the inherently hierarchical and system-level nature of real-world RTL design and leave existing pipelines fragile in multi-module synthesis scenarios, often struggling to converge under verification [17], [18].

In this paper, we introduce Multi-Agent RTL Synthesis (MARS), a novel framework that mirrors the hierarchical, verification-driven workflows used by human hardware designers. MARS addresses the limitations of existing LLM-based system-centric framework approaches by decomposing the RTL synthesis task into a pipeline of specialized agents that coordinate ambiguity resolution, design complexity analysis, modular decomposition, iterative generation, and integration. Incorporating early-stage semantic clarification and complexity-aware routing also enables MARS to dynamically adapt to the nature of the specification—avoiding unnecessary decomposition for simple designs while allowing robust multi-module synthesis when required. Through targeted simulation-based feedback loops and interface-aware verification, the framework achieves significant improvements in functional correctness on established RTL benchmarks.

Our key contributions are as follows:

- We propose a Multi-agent RTL synthesis framework (MARS) that aligns with actual RTL engineering workflows, incorporating hierarchical, context-aware modular decomposition and integration.
- We introduce an early-stage ambiguity resolver and a complexity-aware routing (III-B) mechanism that classifies design tasks and adapts the generation pathway based on structural and behavioral metrics.
- We integrate simulation-guided refinement at the module level and verify structural coherence at the system level, resulting in improved robustness for RTL designs.
- We evaluate MARS on two benchmark suites IV, demonstrating a 5.4% improvement in Pass@1 functional correctness over prior state-of-the-art baselines.

The remainder of this paper is organized as follows. Section II reviews existing approaches to LLM-based RTL synthesis across both model-centric and system-centric axes, and briefly motivates the need for such a framework, as realized by MARS. Section III presents the MARS framework, detailing the design, coordination, and specialized roles of its five agents. Section IV describes our experimental setup, benchmark selection, and comparative evaluation against the state-of-the-art baseline. Finally, Section V summarizes our findings.

II. Related Work and Motivation

The automation of hardware design using LLMs has progressed from simple code completion tasks to complete system-level synthesis. Approaches can be broadly categorized into two directions: model-centric improvements and system-centric framework design.

A. Model-centric Improvements

Early research efforts focused on adapting LLMs to HDLs through domain-specific fine-tuning. **Verigen** [5] demonstrated that fine-tuning open-source LLMs on curated Verilog datasets from github and textbooks significantly improves syntactic correctness and benchmark performance compared to general-purpose LLMs. Similarly, **RTLCoder** [11] proposed a lightweight, fine-tuned open-source model. **CodeV** [19] proposed a "multi-level summarization" data synthesis process to generate high-quality instruction-tuning datasets. This approach enabled the model to handle both Verilog and Chisel. Similarly, **ChipNeMo** [20] utilized domain-adaptive pre-training on domain-specific data to enhance industrial chip design tasks.

While these fine-tuned models excel at generating module-level code and adhering to syntax constraints, they operate as "flat" generators. They often lack the reasoning depth required for system-level architecture, struggling with long-range dependencies and complex state logic when used in a single-pass inference [16].

B. System-centric Frameworks

To overcome the limitations of monolithic generation, recent works have explored frameworks on iterative refinement and agentic collaboration.

1) Iterative Feedback Frameworks: Given the stochastic nature of LLMs, several works have emphasized the importance of simulation-based feedback. Early frameworks such as **AutoChip** [14] adopted a "generate-simulate-repair" loop in which simulation errors are fed back to the model to iteratively correct faulty RTL. **VeriAssist** [21] advanced this concept by introducing a "self-verification" step, prompting the LLM to generate a test plan and reason through execution traces before generating the final code thereby reducing the dependency on external feedback loops alone.

2) Agentic Frameworks: More recently, **ROME** [4] proposed a hierarchical prompting strategy to automatically decompose complex specifications into smaller components. While this method reduces context pressure during generation, it does not explicitly address monolithic RTL generation. **Chip-Chat** [6] demonstrated the potential of conversational interfaces for designing microprocessors, though it relied heavily on human-in-the-loop guidance.

MAGE [16] proposed the first open-source multi-agent framework, assigning distinct roles (RTL generator, Testbench Generator, Simulation Judge) to specialized agents. By separating creator and verifier roles, MAGE significantly improved robustness against self-reinforcing errors. Similarly,

Spec2RTL-Agent [22] mimics human comprehension by utilizing specific agents for understanding, coding, and reflecting on unstructured specification documents, bridging the gap between raw text and synthesizable code.

Beyond HDL synthesis, recent studies have explored integrating LLMs into broader EDA workflows, including script generation for commercial toolchains, debugging and summarization of tool logs, and natural-language interfaces to automate synthesis and physical design [20]. Recent surveys [23] highlight a growing literature applying LLMs across the EDA stack, spanning logic design, verification, and flow automation. In this paper, we focus specifically on RTL synthesis, where correctness and cross-module consistency remain central bottlenecks and motivate system-level LLM frameworks such as MARS.

C. Motivation

Despite these advancements, existing frameworks exhibit two critical limitations that MARS addresses:

1) Monolithic Generation: Existing multi-agent systems [16] [22] adopt a monolithic, flat workflow and follow a "one-size-fits-all" pipeline, applying the same agentic process to both simple designs, such as multiplexers, and complex systems, such as processors.

2) Disambiguation: Existing frameworks operate under the assumption that the specification contains all necessary details. If a specification is fundamentally incomplete (e.g., missing signal widths or protocol definitions) or not clear, these frameworks may hallucinate defaults or ultimately fail.

III. Methodology

The MARS framework is designed to mirror the standard hierarchical workflow of human hardware engineering teams, transitioning from abstract natural language specification to fully verifiable Register Transfer Level (RTL) designs. The architecture operates as a multi-agent system, where distinct LLM-based agents specialize in specification refinement, complexity-aware routing, architectural planning, code generation and system integration. This approach addresses the limitations of monolithic synthesis by ensuring that complex designs are treated as a composition of manageable and verifiable sub-tasks. Fig. 1 illustrates the pipeline consisting of five primary agents:

1) Ambiguity Resolver III-A: Refines under-specifications.
2) Design Complexity Analyzer III-B: Scores specification complexity and determines routing.
3) Design Decomposer III-C: Produces an explicit module-level decomposition with interfaces and interconnections.
4) Module Generator III-D: Generates verifiable modules/sub-modules required for the specification.
5) Integration Agent III-E: Synthesizes a structurally consistent top-level design with generated sub-modules, interfaces and interconnections.

A. Ambiguity Resolver Agent

The RTL synthesis pipeline initiates with the Ambiguity Resolver agent. Human-provided natural language specifications often contain ambiguities, implicit assumptions, or incomplete logic that could lead to hallucinations during code generation by LLMs. The Ambiguity Resolver agent addresses these issues by performing two functions: 1) ambiguity identification and prioritization, and 2) interactive clarification and updating specification. The agent acts as a semantic bridge between human intent and hardware requirements.

Given an initial specification, the agent produces a ranked list of ambiguity items, if the specification is ambiguous. Each item includes a description and a targeted clarification question (optionally accompanied by response options). Ambiguities are sorted by priority to ensure that the most impactful uncertainties are resolved first.

When user responses are available, the agent rewrites the specification into a structured technical description and incorporates the clarified requirements ensuring that subsequent agents operate on a structured, explicitly defined, and unambiguous specification. Fig. 2 shows an ambiguity-resolution artifact produced by MARS, demonstrating the transformation from an initial underspecified prompt to a clarified, implementation-ready specification.

B. Design Complexity Analyzer Agent

Following the refinement of the specification, MARS performs complexity-aware routing through a Design Complexity Analyzer agent which determines whether a design is simple enough for direct RTL synthesis or instead requires a hierarchical decomposition-and-integration pipeline.

The agent accepts the disambiguated specification and subjects it to a quantitative assessment against distinct hardware complexity metrics including:

- **State Machine Complexity**: Evaluates the density of states, transitions, and hierarchical state management.
- **Interface Complexity**: Analyzes the volume of I/O ports, asynchronous boundaries, and cross-domain clocking.
- **Algorithmic Complexity**: Determines the need for dedicated computational units (e.g., DSPs, floating-point logic) versus simple control logic.
- **Structural Complexity**: Assesses the depth of the logic tree and internal register requirements.

The agent assigns a complexity score $S \in [1, 10]$ to each of the metrics. The agent also provides an overall complexity score ($S_{overall}$) which is expected to be the mean of the metrics.

To improve robustness MARS recomputes the overall complexity score (S_{computed}) from the analyzer outputs. If the analyzer-reported score deviates beyond a tolerance bound (δ), the recomputed score is used for downstream routing decisions.

$$S_{final} = \begin{cases} S_{overall}, & \text{if} \quad |S_{computed} - S_{overall}| \leq \delta \\ S_{computed}, & \text{if} \quad |S_{computed} - S_{overall}| > \delta \end{cases}$$

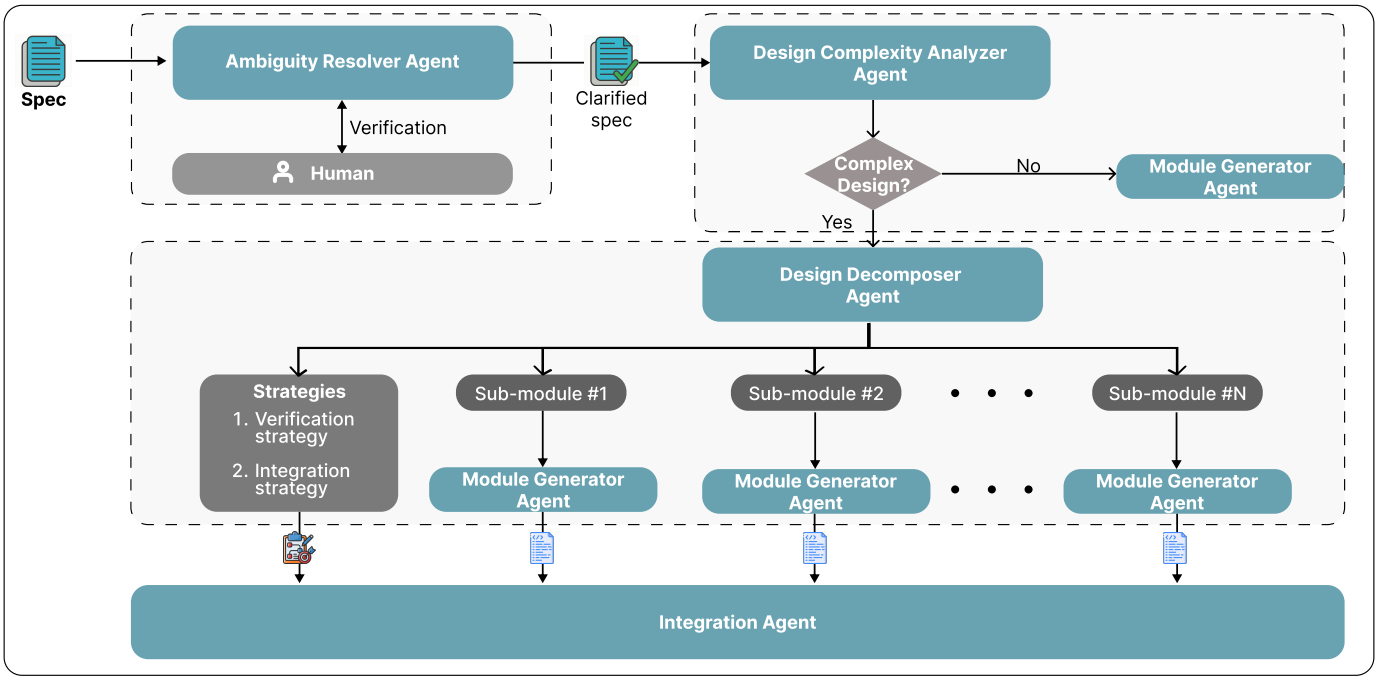

Fig. 1. The MARS workflow

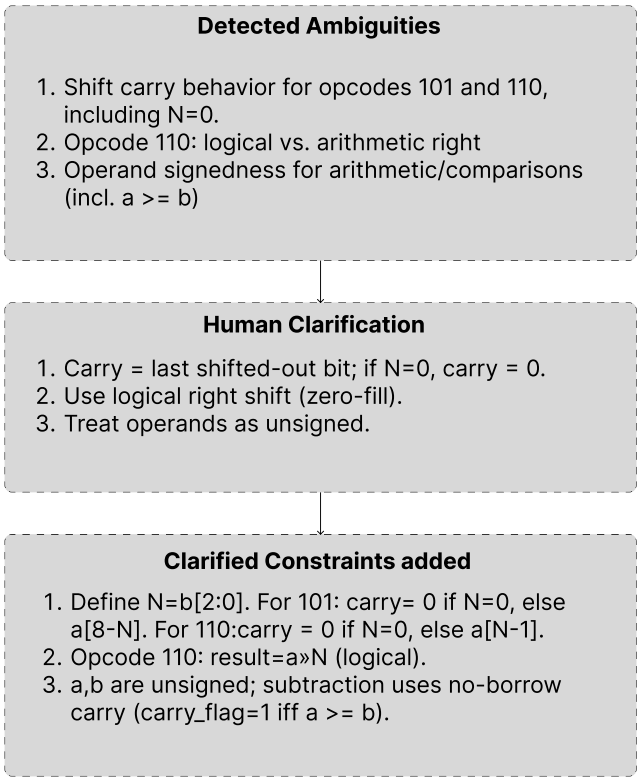

Fig. 2. Ambiguity resolution example

We use a tunable threshold τ (default $\tau = 5.0$) as a conservative midpoint decision boundary to classify the specifications.

$$\text{Classification} = \begin{cases} \text{Simple}, & \text{if} \quad S_{final} < \tau \\ \text{Complex}, & \text{if} \quad S_{final} \geq \tau \end{cases}$$

We do not tune τ to the evaluation set; instead, it serves as a stable default routing rule. Future work can calibrate τ for a given LLM (including fine-tuned variants) to better match model-specific capabilities. Table I shows an example structured report produced for a UART controller specification. The analyzer assigns per-metric scores and computes an overall complexity score S_{overall} used to route the design either to direct RTL generation or hierarchical decomposition.

TABLE I
EXAMPLE COMPLEXITY ANALYSIS OUTPUT FOR A UART CONTROLLER SPECIFICATION

Complexity Metric	Score (1–10)
State Machine Complexity	6.0
Interface Complexity	5.0
Algorithmic Complexity	6.0
Structural Complexity	7.0
Overall score S_{overall}	6.0
Classification	**Complex**

C. Design Decomposer Agent

For specifications deemed complex, the Design Decomposer agent mimics the role of a system architect. It is responsible for partitioning the requirement into a set of discrete, independently verifiable sub-modules. This agent generates a structural plan that defines:

1) Functionality of each sub-module.
2) Interface definition (I/O ports) for each sub-module.
3) Sub-modules dependency indicating how modules interact.

Each sub-module is described by a name, a functional specification, and a structured interface comprising inputs, outputs, clocks, and resets. MARS also extracts explicit dependencies between sub-modules and defines interconnection edges of the form `from_module/from_signal` $\rightarrow$ `to_module/to_signal`. In addition, the structural plan captures global design metadata such as design-level state definitions and global parameters, as well as recommended strategies for integration and verification.

To reduce failure modes stemming from ill-formed decomposition outputs, the decomposer stage includes a post-generation validation pass that checks for common structural inconsistencies, including empty module sets, simple circular dependency patterns, and interconnections referencing undefined modules. The validator also applies naming-based heuristics to identify modules that are likely sequential (e.g., counters, controllers, FSMs) and flags missing clock/reset definitions in their interfaces. These checks provide early detection of decomposition inconsistencies before code generation and integration proceed. Table II presents a decomposition output artifact produced by MARS for a UART controller specification, including the resulting module partition, interface summary, interconnections, and associated verification and integration strategies.

TABLE II
DECOMPOSITION OUTPUT ARTIFACT SNAPSHOT (UART CONTROLLER EXAMPLE)

Output Component	**Extracted Content (Summary)**
Sub-modules	`UartRegsBus`, `BaudGen16x`, `UartTxEngine`, `UartRxEngine`, `SyncFifo8x4` (TX/RX instances)
Top-level I/O	CPU: `address[3:0]`, `data_in[7:0]`, `read`, `write`, `chip_select`, `data_out[7:0]`; UART: `rx`, `tx`; Status: `rx_ready`, `tx_ready`
Clock/reset	Single clock domain: `clk` (50 MHz); synchronous active-low reset `reset_n`
Key interconnects	`baud_sel` drives baud generator; `oversample_tick` shared by TX/RX; TX/RX paths connected via FIFO `push/pop` handshakes; sticky error pulses propagated to CSR block
Global parameters	`SYS_CLK_FREQ`, `OSR`, `DATA_WIDTH`, `FIFO_DEPTH`, baud divisors for {9600, 19200, 38400, 115200}
Verification strategy	Module-first verification (FIFO, baud generator, TX/RX engines, register block), followed by subsystem tests and full integration validation
Integration strategy	Instantiate one register/bus block, one baud generator, one TX engine, one RX engine, and two FIFO instances (`tx_fifo`, `rx_fifo`). Connect CPU bus to `UartRegsBus`, share `oversample_tick`, wire FIFO push/pop paths, export `tx/rx` and ready flags at top-level

D. Module Generation and Verification Agent

The core synthesis engine is the Module Generation agent, which transforms the planned sub-modules (or simple specifications) into functional SystemVerilog code.

To ensure functional correctness, we adapt the iterative feedback architecture introduced in MAGE and optimize it to better handle multiple sub-modules planned by the Design Decomposer agent.

E. Integration Agent

Once all sub-modules have successfully passed their individual verification, the Integration Agent is invoked. This agent synthesizes the top-level module by instantiating sub-modules, declaring internal wiring, and mapping ports according to the decomposition output. To reduce integration-time errors, the agent additionally produces structured compatibility checks for each interconnection.

IV. EXPERIMENTS AND RESULTS

A. Experimental Setup

We evaluate the proposed MARS framework on two established hardware description benchmarks: VerilogEval-V1 [24] and CVDP [18], which collectively cover a broad spectrum of combinational and sequential logic design tasks. The evaluation set consists of 74 problems in total, comprising 20 problems from VerilogEval-V1 and 54 problems from CVDP. To avoid selection bias, the problems were randomly sampled from the dataset.

For each problem, the generated Register Transfer Level (RTL) code is validated for functional correctness using Icarus Verilog [25], an open-source Verilog compiler and simulator. A solution is considered correct only if it successfully compiles and passes all provided test vectors.

B. Model Configuration and Baseline

The core reasoning engine used by MARS is GPT-5 [26], accessed through the LlamaIndex orchestration framework. The model is configured with the Medium Reasoning preset.

We compare MARS against MAGE, a state-of-the-art open-source framework for RTL generation. To ensure a strictly controlled comparison, MAGE is re-implemented using the same backbone model (GPT-5 [26]), identical reasoning configuration, and the same verification environment. This experimental design isolates performance differences attributable solely to architectural choices rather than underlying model capability.

C. Evaluation Metric

Following the recent works [16] [21] [27], performance is measured using the Pass@1 metric [28].

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

where n is the total number of problems, c is the number of problems passed, k = 1. We evaluate the single, final solution produced by the framework for each problem. The metric

reflects the framework's ability to yield a correct design in a single invocation.

D. Results

Table III summarizes the Pass@1 performance of MARS and the MAGE baseline across the combined benchmark suite using GPT-5 [26] as the base model.

TABLE III
PASS@1 PERFORMANCE COMPARISON

Framework	Pass@1 (%)
MAGE (Baseline)	75.68
MARS (Ours)	**81.08**

MARS achieves an aggregate Pass@1 rate of 81.08%, outperforming the MAGE baseline by 5.4 percentage points.

The observed performance gains are largely attributable to the MARS architecture. The key architectural factors contributing to this superior performance are:

- **Decomposition and Integration pipeline**: Complex specifications are decomposed into smaller submodules and subsequently integrated through well-defined interfaces and interconnections.
- **Preprocessing**: Specialized agents for ambiguity resolution and complexity analysis improve the quality of downstream code generation by clarifying underspecified requirements and selecting an appropriate synthesis route.

V. CONCLUSION

In this paper, we introduced **MARS**, a multi-agent RTL synthesis framework that overcomes the limitations of treating the specification as a flat and monolithic unit by mirroring the hierarchical workflow of human hardware engineering teams.

By decoupling the design process into distinct stages of disambiguation, complexity analysis, modular decomposition, per-module RTL generation and integration, MARS transforms the stochastic nature of LLM-based generation into a more structured and repeatable engineering pipeline. The Ambiguity Resolver acts as a semantic validation layer, ensuring that downstream agents operate on a structured and unambiguous specification, while the Complexity Analyzer determines whether hierarchical decomposition is necessary based on design complexity. Furthermore, by integrating iterative verification loops within a structured decomposition-and-integration architecture, MARS demonstrates that partitioning complex system-level designs into granular, independently verifiable sub-modules improves synthesis reliability.

ACKNOWLEDGEMENTS

This work was supported in part by NYUAD Center for Cyber Security (CCS), funded by Tamkeen under the NYUAD Research Institute Award G1104.

REFERENCES

[1] R. Nane *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.
[2] J. Blocklove *et al.*, "Evaluating llms for hardware design and test," in *2024 IEEE LLM Aided Design Workshop (LAD)*, pp. 1–6, IEEE, 2024.
[3] M. Shao *et al.*, "Survey of different large language model architectures: Trends, benchmarks, and challenges," *IEEE Access*, 2024.
[4] A. Nakkab *et al.*, "Rome was not built in a single step: Hierarchical prompting for llm-based chip design," 2024.
[5] S. Thakur, B. Ahmad, H. Pearce, B. Tan, B. Dolan-Gavitt, R. Karri, and S. Garg, "Verigen: A large language model for verilog code generation," 2023.
[6] J. Blocklove, S. Garg, R. Karri, and H. Pearce, "Chip-chat: Challenges and opportunities in conversational hardware design," in *2023 ACM/IEEE 5th Workshop on Machine Learning for CAD (MLCAD)*, p. 1–6, IEEE, Sept. 2023.
[7] Y. Lu *et al.*, "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)*, pp. 722–727, IEEE, 2024.
[8] Z. Wang *et al.*, "Veridispatcher: Multi-model dispatching through pre-inference difficulty prediction for rtl generation optimization," *arXiv preprint arXiv:2511.22749*, 2025.
[9] S. Thakur *et al.*, "Benchmarking large language models for automated verilog rtl code generation," 2022.
[10] N. Wang, B. Yao, J. Zhou, X. Wang, Z. Jiang, and N. Guan, "Large language model for verilog generation with code-structure-guided reinforcement learning, 2025," *URL https://arxiv. org/abs/2407.18271*.
[11] S. Liu, W. Fang, Y. Lu, Q. Zhang, H. Zhang, and Z. Xie, "Rtlcoder: Outperforming gpt-3.5 in design rtl generation with our open-source dataset and lightweight solution," in *2024 IEEE LLM Aided Design Workshop (LAD)*, p. 1–5, IEEE, June 2024.
[12] W. Xiao *et al.*, "Trojanloc: Llm-based framework for rtl trojan localization," *arXiv preprint arXiv:2512.00591*, 2025.
[13] B. Mali, K. Maddala, V. Gupta, S. Reddy, C. Karfa, and R. Karri, "Chiraag: Chatgpt informed rapid and automated assertion generation," in *2024 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, pp. 680–683, IEEE, 2024.
[14] S. Thakur, J. Blocklove, H. Pearce, B. Tan, S. Garg, and R. Karri, "Autochip: Automating hdl generation using llm feedback," 2024.
[15] Z. Wang *et al.*, "Netdetox: Adversarial and efficient evasion of hardware-security gnns via rl-llm orchestration," *arXiv preprint arXiv:2512.00119*, 2025.
[16] Y. Zhao, H. Zhang, H. Huang, Z. Yu, and J. Zhao, "Mage: A multi-agent engine for automated rtl code generation," 2024.
[17] R. Moundir Ghorab, E. Parisi, C. Gutierrez, M. Alberti-Binimelis, M. Moreto, D. Garcia-Gasulla, and G. Kestor, "Notsotiny: A large, living benchmark for rtl code generation," *arXiv e-prints*, pp. arXiv–2512, 2025.
[18] N. Pinckney *et al.*, "Comprehensive verilog design problems: A next-generation benchmark dataset for evaluating large language models and agents on rtl design and verification," 2025.
[19] Y. Zhao *et al.*, "Codev: Empowering llms with hdl generation through multi-level summarization," 2025.
[20] M. Liu *et al.*, "Chipnemo: Domain-adapted llms for chip design," 2024.
[21] H. Huang *et al.*, "Towards llm-powered verilog rtl assistant: Self-verification and self-correction," 2024.
[22] Z. Yu, M. Liu, M. Zimmer, Y. C. Lin, Y. Liu, and H. Ren, "Spec2rtl-agent: Automated hardware code generation from complex specifications using llm agent systems," 2025.
[23] J. Pan, G. Zhou, C.-C. Chang, I. Jacobson, J. Hu, and Y. Chen, "A survey of research in large language models for electronic design automation," 2025.
[24] M. Liu, N. Pinckney, B. Khailany, and H. Ren, "Verilogeval: Evaluating large language models for verilog code generation," 2023.
[25] S. Williams, "The icarus verilog compilation system."
[26] OpenAI, "Gpt-5 is here — openai."
[27] M. Gao *et al.*, "Autovcoder: A systematic framework for automated verilog code generation using llms," 2024.
[28] M. Chen *et al.*, "Evaluating large language models trained on code," 2021.