CODMAS: A Dialectic Multi-Agent Collaborative Framework for Structured RTL Optimization

Che-Ming Chang[*1], **Prashanth Vijayaraghavan**[*2], **Ashutosh Jadhav**[2], **Charles Mackin**[2], **Vandana Mukherjee**[2], **Hsinyu Tsai**[2], **Ehsan Degan**[2]

[1]National Taiwan University, [2]IBM Research
b09901156@ntu.edu.tw, prashanthv@ibm.com, ashutosh@us.ibm.com,
charles.mackin@ibm.com, vandana@us.ibm.com,
htsai@us.ibm.com, edehgha@us.ibm.com

Abstract

Optimizing Register Transfer Level (RTL) code is a critical step in Electronic Design Automation (EDA) for improving power, performance, and area (PPA). We present CODMAS (*Collaborative Optimization via a Dialectic Multi-Agent System*), a framework that combines structured dialectic reasoning with domain-aware code generation and deterministic evaluation to automate RTL optimization. At the core of CODMAS are two dialectic agents: the *Articulator*, inspired by rubber-duck debugging, which articulates stepwise transformation plans and exposes latent assumptions; and the *Hypothesis Partner*, which predicts outcomes and reconciles deviations between expected and actual behavior to guide targeted refinements. These agents direct a Domain-Specific Coding Agent (DCA) to generate architecture-aware Verilog edits and a Code Evaluation Agent (CEA) to verify syntax, functionality, and PPA metrics. We introduce RTLOPT, a benchmark of 120 Verilog triples (unoptimized, optimized, testbench) for pipelining and clock-gating transformations. Across proprietary and open LLMs, CODMAS achieves ∼25% reduction in critical path delay for pipelining and ∼22% power reduction for clock gating, while reducing functional and compilation failures compared to strong prompting and agentic baselines. These results demonstrate that structured multi-agent reasoning can significantly enhance automated RTL optimization and scale to more complex designs and broader optimization tasks.

1 Introduction

The increasing complexity of modern chip design has accelerated the integration of Artificial Intelligence (AI) into Electronic Design Automation (EDA) workflows, reducing reliance on manual effort and enabling faster design cycles. While AI has shown notable success in tasks such as logic synthesis (Pei et al., 2023) and placement (Lai et al., 2023), generating and optimizing Hardware Description Languages (HDLs) remains challenging. RTL (Register Transfer Level) code, typically written in Verilog or VHDL, requires careful hand-tuned transformations to meet power, performance, and area (PPA) constraints. Transformations such as pipelining and clock gating are particularly critical for performance and energy efficiency, yet they are time-consuming, error-prone, and demand deep domain expertise. Commercial EDA tools provide automated support for certain RTL optimizations, but they lack explicit reasoning about design intent and early-stage architectural transformations. Existing learning-based approaches either focus on syntactic HDL generation (Thakur et al., 2024a; Akyash et al., 2025; Yu et al., 2025; Liu et al., 2024b; Chang et al., 2023; Thakur et al., 2024b; Ho et al., 2024) or rely on heuristic search or local rewriting (Thorat et al., 2024; Yao et al., 2024; DeLorenzo et al., 2024), limiting their ability to generalize across designs and capture global architectural patterns. Moreover, datasets for RTL optimization remain scarce, limiting reproducibility, benchmarking, and systematic evaluation of learning-based methods.

To address these gaps, we introduce RTLOPT, a

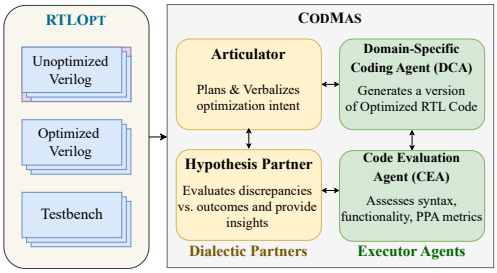

Figure 1: Overview of the CODMAS framework, illustrating dialectic interaction between agents and iterative refinement of RTL code.

[*]Both authors contributed equally to this work. This work was conducted in part during an internship at IBM Research.

Proceedings of the 19th Conference of the European Chapter of the Association for Computational Linguistics (Volume 5: Industry Track), pages 777–788

curated benchmark designed specifically for RTL-level optimization. It contains 120 Verilog code triples, each consisting of an unoptimized design, an optimized counterpart (via pipelining or clock gating), and an associated testbench. This organization enables both functional verification and quantitative evaluation of PPA metrics. The dataset spans diverse design patterns and complexity levels, providing a tractable yet representative foundation for research-scale experimentation and model evaluation. Building on this benchmark, we propose CODMAS, a multi-agent framework for automated RTL optimization that integrates *dialectic reasoning* into the optimization loop. The *Articulator* agent verbalizes optimization intent, while the *Hypothesis Partner* evaluates discrepancies between expected and actual outcomes. These reasoning agents coordinate with the *Domain-Specific Coding Agent (DCA)* and the *Code Evaluation Agent (CEA)* to iteratively refine RTL designs, maintaining functional correctness while improving PPA metrics. Figure 1 illustrates this closed-loop interaction. Our key contributions include:

RTLOPT: A benchmark optimization dataset of ∼120 Verilog-based pipelining and clock gating.

CODMAS: A multi-agent framework combining dialectic reasoning, domain-informed code generation, and PPA evaluation for RTL optimization.

Empirical validation: Demonstrates consistent improvements across models and optimization scenarios, highlighting the efficacy of structured reasoning in automated RTL optimization.

2 RTLOPT: HDL Optimization Dataset

To evaluate LLM performance in HDL code optimization, we collected data from GitHub repositories implementing optimization methods, focusing on Verilog. Since techniques like pipelining and clock gating are not always explicitly labeled, we used search terms like "(pipelining OR clock gating) + verilog" (e.g., "pipelining verilog" or "pipelined verilog") to identify relevant repositories. The data collection process was as follows: we filtered all publicly available repositories for self-contained Verilog files to reduce cross-file dependencies and simplify analysis. We then extracted accompanying testbenches and performed deduplication on the collected modules using token-level similarity and AST-structure hashing to reduce near-duplicates and mitigate overlap with public training corpora.

RTLOPT Dataset Statistics	
# Pipelining Code Triples	70
# Clock Gating Code Triples	50
Power Range (nW)	$\sim 1- \sim 19,000$
Area Range (μm^2)	$\sim 1- \sim 18,000$
Delay Range (ns)	$\sim 15- \sim 1,600$

Table 1: Summary of the RTLOPT Dataset for evaluating pipelining and clock gating optimizations.

To specifically target pipelining and clock gating optimizations, we manually reviewed filtered Verilog files, inspecting code or descriptions indicating these optimization patterns. When optimization details were missing, we generated unoptimized versions (i.e., without pipelining or clock gating) and corresponding testbenches, either by modifying existing descriptions or editing LLM-generated code (e.g., DeepSeek models (Liu et al., 2024a)). All generated or edited examples were subsequently validated through synthesis and simulation to ensure functional equivalence and measurable metric improvements, and were normalized to minimize stylistic cues between unoptimized and optimized pairs. Importantly, unoptimized versions were obtained independently rather than mechanically degraded from optimized code, avoiding reverse-engineering artifacts.

This process resulted in the RTLOPT dataset, consisting of triples of (unoptimized code, optimized code, testbench) for evaluation with designs spanning arithmetic, control, and mixed modules, ensuring diversity. While modest in scale (120 triples), RTLOPT is comparable to or larger than existing code evaluation benchmarks and is designed to prioritize realistic transformations and reproducible evaluation over raw size. The current form of the dataset focuses on pipelining and clock gating, but the dataset structure and tooling are designed to support future extensions to transformations such as retiming, resource sharing, and FSM restructuring. RTLOPT is a seed benchmark intended for community extension. See Table 1 for dataset statistics.

3 Methodology

We introduce CODMAS (*Collaborative Optimization via Dialectic Multi-Agent System for Structured RTL*), a multi-agent framework for automated RTL optimization. CODMAS is grounded on two insights: (1) iterative, structured reasoning between complementary agents enhances optimiza-

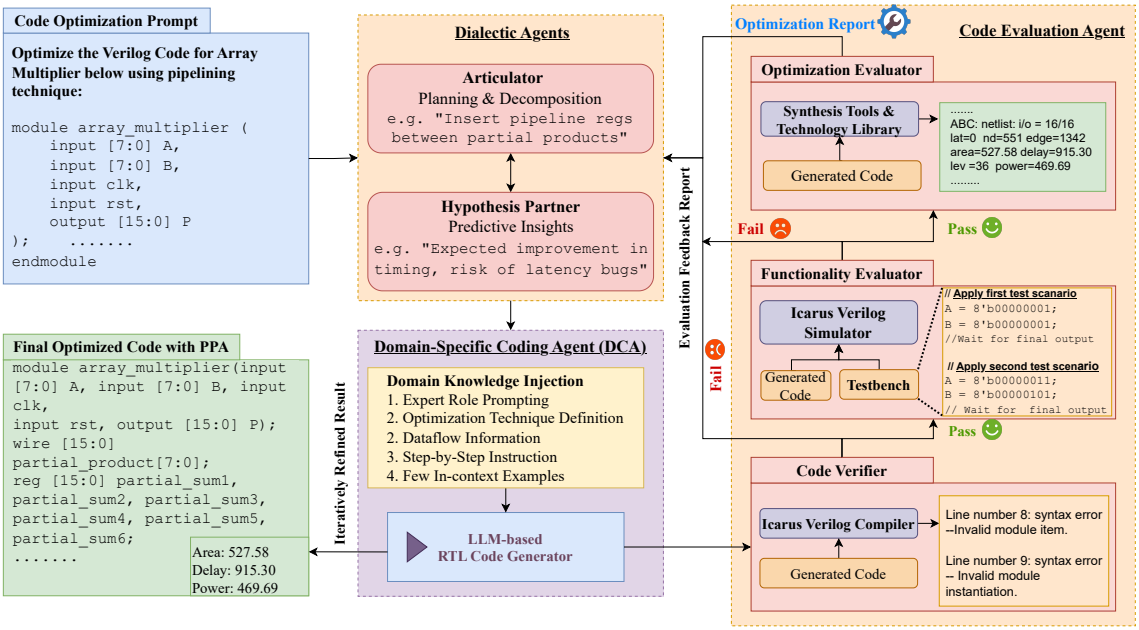

Figure 2: Illustration of the complete CODMAS architecture, showing dialectic agents (Articulator & Hypothesis Partner), the executor agents (Domain-Specific Coding Agent & Code Evaluation Agent), and the iterative RTL refinement loop (e.g., pipelined array multiplier). The example highlights a pipelined array multiplier optimization, with feedback from different components guiding successive refinements toward improved PPA metrics.

tion quality, and (2) domain-specific architectural knowledge must guide LLM-based RTL transformations. The system comprises four specialized agents: two *dialectic reasoning agents* (Articulator and Hypothesis Partner) and two *executor agents* (Domain-Specific Coding Agent (DCA), and Code Evaluation Agent (CEA)). Unlike conventional LLM-based pipelines, CODMAS integrates planning, hypothesis formation, code generation, and deterministic evaluation in a closed feedback loop. This structure supports interpretable, performance-aware refinement and can generalize beyond the initial 120 Verilog triples in RTLOPT, allowing application to larger designs, cross-file modules, and diverse RTL transformations.

3.1 Dialectic Reasoning Agents

The dialectic reasoning agents in **CODMAS** operate as complementary collaborators within a structured, pair programming-inspired optimization workflow. Unlike typical single-agent or monolithic reasoning pipelines, RTL optimization benefits from explicitly separating *design articulation* from *hypothesis generation*. Accordingly, both agents engage in reflective design thinking but assume distinct cognitive roles: the **Articulator** focuses on decomposition and planning, while the **Hypothesis Partner** specializes in predictive reasoning and diagnostic inference. Their iterative interaction forms a closed-loop dialectic, where articulation, critique, and hypothesis refinement guide the executor agents toward functionally correct and performance-optimized RTL.

3.1.1 Articulator

The Articulator serves as the planning and verbalization module of the system. Inspired by *rubber-duck debugging* and structured reasoning, it analyzes the unoptimized RTL design then generates a stepwise transformation plan aligned with target optimization objectives, such as pipelining or clock gating. Each step is interpretable and designed to preserve functional correctness, enabling traceable optimization workflows. Procedural reasoning allows the Articulator to decompose complex transformations into ordered operations while surfacing latent design assumptions. These assumptions are expressed in a structured form consumable by both human designers and executor agents. For instance, it may recommend inserting pipeline registers between critical computation stages or adjusting delay alignment logic to preserve correctness. This articulated plan provides a shared semantic scaffold for hypothesis formation, code generation, and iterative refinement, remaining largely stable across hypothesis proposals to enable consistent evaluation of alternative transformations under a unified circuit model.

3.1.2 Hypothesis Partner

The Hypothesis Partner operates in tandem with the Articulator, leveraging *abductive* and *model-based reasoning* to anticipate the effects of planned transformations on functional correctness and PPA metrics. Unlike conventional code-generation approaches, it begins reasoning before any optimized code is produced, formulating hypotheses conditioned on the Articulator's performance-annotated circuit representation. Given a fixed structural interpretation, the Hypothesis Partner proposes metric-targeted transformations, predicts performance gains, identifies functional or structural pitfalls, and guides corrective strategies. When synthesis or simulation feedback reveals deviations, the agent revises prior hypotheses, attributes failure causes, and suggests targeted refinements. The iterative reasoning loop embeds verification-awareness into the optimization process, ensuring that code generation is guided by actionable, performance-aligned insights. This separation enables systematic exploration of multiple candidate transformations under a shared design model, avoiding premature commitment to suboptimal directions. As shown in Section 5.2, collapsing these roles into a unified agent degrades convergence stability and final metric gains, supporting the necessity of this dialectic structure for RTL optimization.

3.2 Executor Agents

Executor agents in **CODMAS** operationalize the dialectic agents' reasoning by generating candidate RTL and providing rigorous feedback. While dialectic agents handle planning and predictive inference, executor agents convert these insights into actionable code and deterministic evaluation. This layer comprises the *Domain-Specific Coding Agent (DCA)* for optimized RTL generation and the *Code Evaluation Agent (CEA)* for syntax, functional, and PPA assessment. Together, they form a tightly coupled loop that iteratively refines RTL designs.

3.2.1 Domain-Specific Coding Agent (DCA)

The DCA translates the Articulator's transformation plan and the Hypothesis Partner's predicted outcomes into first-pass optimized Verilog code. It employs a multi-faceted prompt strategy via the *Domain Knowledge Injector (DKI)*: (a) it positions the LLM as an expert RTL designer instructed to apply optimization techniques such as pipelining or clock gating (*role prompting*); (b) embeds K annotated examples of unoptimized and optimized Verilog code aligned with the articulated plan and predicted outcomes (*example-based guidance*); and (c) incorporates structural context from a dataflow graph extracted with Pyverilog (*tool-informed context*). Rather than providing raw RTL alone, the injected dataflow graph encodes register stages, combinational logic cones, control dependencies, and clock enables in a constrained schema, enabling the model to reason explicitly about pipeline depth, stage imbalance, and safe gating opportunities. For example, in a pipelining task, the graph exposes a long combinational path between two registers, guiding the insertion of intermediate registers while preserving control alignment.

Using this enriched prompt, the DCA generates a functionally equivalent, architecture-aware candidate design. The CEA evaluates this code, and feedback is returned to the dialectic agents: the Articulator revises transformation steps, the Hypothesis Partner updates predicted outcomes, and the DCA incorporates these adjustments into the next iteration. This cycle repeats until functional correctness and target PPA improvements are achieved.

3.2.2 Code Evaluation Agent (CEA)

The CEA provides deterministic, verifiable evaluation using open-source EDA tools (Icarus Verilog for compilation, GTKWave for simulation, and Yosys for synthesis and timing analysis). It evaluates designs along three dimensions: syntactic correctness, functional equivalence, and optimization quality. The *Code Verifier* ensures compilable Verilog and produces structured error messages to guide correction; the *Functionality Evaluator* simulates designs with testbenches adjusted for transformations such as pipelining to detect behavioral mismatches or regressions; and the *Optimization Evaluator* synthesizes designs to extract PPA metrics for performance-aware refinement. The CEA compiles a unified feedback report consumed by the dialectic agents: syntactic errors prompt revision of transformation plans, functional mismatches trigger updates to assumptions, and suboptimal PPA metrics guide optimization strategy adjustments. The updated plans and hypotheses are sent back to the DCA, which generates refined RTL code.

3.3 Integrated Optimization Workflow

Algorithm 1 summarizes the CODMAS optimization pipeline. Given unoptimized RTL C_0, target goals G (e.g., PPA improvements), and iteration limit T, the *Articulator* generates a structured trans-

formation plan while the *Hypothesis Partner* predicts expected functional behavior and PPA outcomes. A dataflow DAG supports structural reasoning, and the *Domain Knowledge Injector* composes a prompt combining plan, hypotheses, and structural context for the LLM to generate candidate optimized RTL. The *Code Evaluation Agent (CEA)*, comprising the Code Verifier, Functionality Evaluator, and Optimization Evaluator, assesses syntax, functional equivalence, and PPA. Feedback is incorporated into the dialectic loop: syntactic errors adjust the plan, functional mismatches update assumptions, and suboptimal PPA metrics guide strategy refinements. This loop continues until optimization goals are met or the iteration limit is reached, providing a traceable, verification-aware, and performance-driven optimization workflow.

4 Experiments

We evaluate CODMAS on RTL optimization, presenting the experimental setup, baselines, and metrics. Our study addresses three questions: **RQ1: Effectiveness of CODMAS:** How does the framework improve PPA compared to other baselines? **RQ2: Component-wise Contribution:** What is the impact of the dialectic and executor agents on optimization performance? **RQ3: Benefits of Iterative Refinement:** How do multi-step reasoning and iterative refinement affect convergence speed and optimization quality across designs?

Baselines We compare CODMAS against representative prompting and agent-based approaches for RTL optimization. Our baselines include **Zero-Shot** prompting, intermediate-reasoning methods such as **CoDes** (Vijayaraghavan et al., 2024), **ReAct** (Yao et al., 2023), and **Reflexion** (Shinn et al., 2024), and the two-stage correction–optimization pipeline **LLM-VeriPPA** (Thorat et al., 2024). All systems are evaluated using identical simulation and synthesis flows across proprietary models (GPT-4o, GPT-3.5-turbo) and open-source models (Llama-3, DeepSeek-v2.5, Granite-34B-Code, CodeLLaMA-34B). Detailed baseline configurations appear in the Appendix D. Although baselines such as ReAct, Reflexion, and CoDes were originally developed for general software reasoning tasks, we adapt them for RTL optimization by providing the same RTL parsing, transformation, and synthesis-feedback interfaces as used in CODMAS. This ensures a fair comparison while preserving the original reasoning strategies of these models within the hardware optimization context.

Metrics We evaluate optimized RTL relative to the original design using synthesis and simulation with Yosys and Liberty-based standard-cell libraries. Area (A) is reported as A/A_0, where A_0 is the baseline, with values below 1 indicating reduction. Power (P) and timing (T) improvements are expressed as percentage gains, with timing measured via critical path delay (CPD), where positive gains indicate reduced CPD. Failure rate (FR) captures the fraction of designs that fail functional, synthesis, or timing checks after optimization, with lower FR indicating higher reliability (see Appendix B). All optimized designs are validated using module-level testbenches derived from original repositories or constructed from behavioral specifications. While we do not yet incorporate formal equivalence checking, our methodology aligns with common industrial RTL optimization pipelines, where simulation-based testing remains one of the key validation mechanisms.

5 Results

5.1 (RQ1) Effectiveness of CODMAS

Table 2 summarizes PPA outcomes for six LLMs under four optimization strategies. Across all models, CODMAS consistently delivers the strongest improvements: pipelining achieves timing gains above 20% (reaching 25.5% on GPT-4o), while clock gating attains power reductions exceeding 19% on average, compared to less than 10% for all baselines. Failure rates under CODMAS remain below 30%, while prompting and agentic baselines typically exceed 40% to 50%, indicating more frequent syntax, functional, and PPA violations. Model-wise trends reinforce these findings. GPT-4o leads across strategies, achieving $\sim 25\%$ timing improvement in pipelining and $\sim 22\%$ power reduction in clock gating with FR below 25%. Open-source models Granite-34b and CodeLlama-34b perform poorly under zero-shot or naive prompting, with minimal PPA gains and FR above 60%, yet under CODMAS they reach up to 13% timing and power improvements with FR under 30%. Llama3 is the most challenging: baseline modes increase area up to 18% with FR above 50%, but CODMAS reduces net area impact to $A/A_0 \approx 1.03$ while achieving timing gains near 20%. Error analysis indicates baseline failures arise from syntax ($\sim 40\%$), functional ($\sim 35\%$), and PPA ($\sim 25\%$) issues,

Pipelining												
Models	**Zero-Shot**			**Prompting/Agentic**			**LLM-VeriPPA**			**CODMAS**		
	A (↓)	T (↑)	FR (↓)	A (↓)	T (↑)	FR (↓)	A (↓)	T (↑)	FR (↓)	A (↓)	T (↑)	FR (↓)
GPT-4o	**0.985**	**11.4**	**54.2**	**0.982**	**15.2**	**49.6**	**0.986**	**15.4**	**39.7**	**0.960**	**25.5**	**19.5**
GPT-3.5-turbo	0.989	10.0	58.0	0.988	14.0	51.2	1.008	13.6	43.3	0.972	21.3	23.4
DeepSeek-v2.5	1.009	9.8	57.8	0.986	12.8	50.0	1.025	13.0	42.8	0.979	21.4	22.8
Llama-3	1.045	8.7	61.4	1.181	11.6	53.8	1.116	11.3	47.1	1.032	19.8	25.5
Granite-34b	**1.026**	3.9	65.2	**1.015**	5.3	**59.1**	**1.022**	6.6	57.7	**0.998**	10.5	**28.3**
CodeLlama-34b	1.035	**4.2**	**64.7**	1.036	**6.8**	60.7	1.039	**7.2**	**56.1**	1.030	**11.2**	29.5
Human	N/A			N/A			N/A			**0.848**	**45.6**	**0.0**
Clock Gating												
Models	**Zero-Shot**			**Prompting/Agentic**			**LLM-VeriPPA**			**CODMAS**		
	A (↓)	P (↑)	FR (↓)	A (↓)	P (↑)	FR (↓)	A (↓)	P (↑)	FR (↓)	A (↓)	P (↑)	FR (↓)
GPT-4o	**1.023**	**7.8**	**52.5**	**1.010**	**9.1**	**46.8**	**1.009**	**9.3**	**38.9**	**0.999**	**21.7**	**21.8**
GPT-3.5-turbo	1.035	6.3	55.6	1.018	7.5	49.6	1.019	7.9	43.4	1.020	18.8	24.2
DeepSeek-v2.5	1.030	6.2	54.8	1.014	7.8	48.9	1.020	8.2	42.9	1.015	19.0	23.6
Llama-3	1.093	5.5	58.6	1.060	6.7	52.1	1.062	7.0	46.5	1.048	16.5	26.3
Granite-34b	**1.037**	3.0	63.0	**1.048**	**5.8**	59.8	**1.049**	**5.9**	**55.8**	**1.030**	**12.9**	**29.5**
CodeLlama-34b	1.054	**3.4**	**61.6**	1.053	5.5	**58.7**	1.050	5.4	55.3	1.042	10.6	31.3
Human	N/A			N/A			N/A			**0.925**	**30.4**	**0.0**

Table 2: Performance comparison on pipelining and clock gating (CG). Pipelining reports Area (A), Timing (T), and Failure Rate (FR); clock gating reports A, Power (P), and FR. Bold indicates best; ↑ higher is better, ↓ lower is better. All improvements are statistically significant ($p < 0.01$) via paired t-tests with standard deviation in Table 6.

all mitigated by CODMAS through explicit planning, hypothesis-guided reasoning, and deterministic evaluation. Overall, CODMAS consistently improves PPA, lowers FR, and stabilizes optimization across both proprietary and open-source LLMs.

5.1.1 Impact on Area

Pipelining adds registers and handshake logic, and clock gating adds gating cells and control logic, often increasing area despite timing or power gains. In our experiments, CODMAS keeps area near baseline, with GPT-4o showing $\sim 4\%$ reductions under both optimizations while achieving notable timing or power improvements. Area changes are generally modest, due to synthesis variations or minor restructuring. Baselines typically show minimal area change with limited PPA gains; for example, Llama-3 in zero-shot pipelining increases area $\sim 4.5\%$ with only $\sim 8.7\%$ timing gain and high FR (61.4%). The Articulator's transformations and Hypothesis Partner's forecasts, validated by the CEA, focus on PPA improvements while keeping area changes secondary.

5.2 (RQ2) Component-wise Contribution

Table 3 presents an ablation study of three key CODMAS components: Dialectic Agents (DA), Domain Knowledge Injection (DKI), and the Code Evaluation Agent (CEA), reporting metrics for GPT-4o and Llama-3. The complete CODMAS pipeline consistently outperforms all ablated variants, demonstrating the importance of each module: DA coordinates structured reasoning, DKI grounds transformations in design intent, and CEA filters invalid or low-quality edits.

Removing the Dialectic Agents (*w/o DA*) results in the largest drop in performance. Timing gains for GPT-4o pipelining fall from 25.5% to 12.9%, and failure rates rise to 38.5%–44.7%, highlighting DA's role in structured refinement and hypothesis-guided filtering. Without CEA (*w/o CEA*), failure rates increase (e.g., 21.8% to 32.7% for GPT-4o clock gating), as flawed edits persist. Omitting DKI (*w/o DKI*) reduces optimization quality and robustness, particularly for smaller models: Llama-3 pipelining timing gains drop from 19.8% to 9.3%, and FR rises by ~ 10 points. These results confirm that each component (DA, DKI, and CEA) provides complementary benefits that are crucial for achieving stable RTL optimization.

5.2.1 Dialectic Agent Ablation

To isolate the impact of the dialectic agent design in CODMAS, we compare the full system against

	FULL	w/o CEA	w/o DA	w/o DKI
Pipelining				
T (GPT-4o)	**25.5**	17.2	12.9	11.5
FR (GPT-4o)	**19.5**	31.0	38.5	32.3
T (Llama-3)	**19.8**	12.6	10.4	9.3
FR (Llama-3)	**25.5**	36.9	44.7	35.8
Clock Gating				
P (GPT-4o)	**21.7**	14.4	9.5	11.1
FR (GPT-4o)	**21.8**	32.7	37.2	40.5
P (Llama-3)	**16.5**	10.2	6.8	7.2
FR (Llama-3)	**26.3**	35.5	41.3	43.2

Table 3: Component-wise ablation study of CODMAS. DA: Dialectic Agents; DKI: Domain Knowledge Injection; CEA: Code Evaluation Agent.

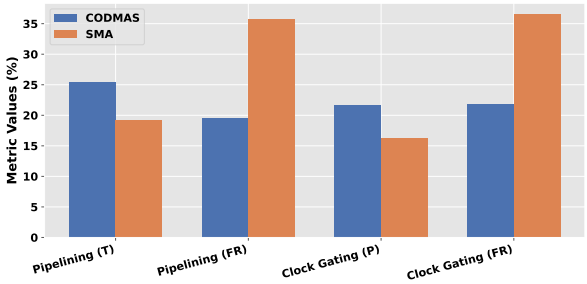

Figure 3: Ablation comparing CODMAS with a shared-memory multi-agent (SMA) variant where the Articulator and Hypothesis Partner roles are combined.

a single alternative architecture: a shared-memory multi-agent (SMA) variant in which the Articulator and Hypothesis Partner roles are collapsed, and both agents jointly interpret and modify the RTL without explicit separation of planning and predictive reasoning. All variants are matched for synthesis calls to ensure fair comparison.

Results in Figure 3 show that collapsing the two roles into a shared-memory multi-agent system consistently reduces performance gains and increases failure rates. For instance, pipelining timing improvements drop from 25.5% to 19.2% for GPT-4o, and failure rates increase from 19.5% to 35.7%. Similarly, clock gating power gains decrease and FR rises. These findings demonstrate that the explicit separation between the Articulator and Hypothesis Partner is critical: articulated planning provides a stable semantic scaffold, while the predictive hypotheses guide targeted transformations. Together, this structure enables higher metric gains, and more reproducible RTL optimization.

5.3 (RQ3) Benefits of Iterative Refinement

To assess the impact of iterative feedback on pipelining, we evaluate *easy* and *hard* tasks over

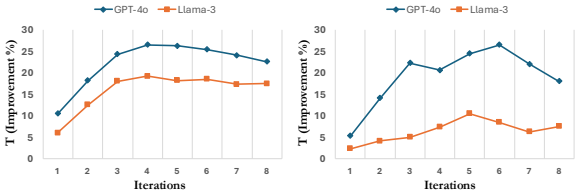

Figure 4: Impact of iterative refinement in CODMAS on pipelining timing improvement (%) for easy (left) and hard (right) RTLOPT problems. Early iterations yield substantial gains; later iterations show diminishing or unstable returns, especially for harder cases.

eight refinement iterations (Figure 4). Both scenarios show steep gains in the first three iterations, indicating that initial feedback captures the largest optimization opportunities. Easy tasks plateau by iteration 4 and decline slightly thereafter, while hard tasks peak around iterations 5-6 and then oscillate, reflecting dense pipelines and conflicting transformation hypotheses. These trends highlight the value of iterative refinement for systematically improving performance, while also indicating that excessive iterations may yield marginal or unstable gains. Adaptive stopping or dynamic iteration strategies can mitigate wasted computation and prevent regressions in complex designs.

6 Conclusion

Achieving efficient power, performance, and area (PPA) in RTL designs is a critical yet challenging task in modern hardware design. To address this, we introduce RTLOPT, a benchmark for pipelining and clock-gating optimizations, and CODMAS, a multi-agent framework combining dialectic reasoning with domain-specific code generation and deterministic evaluation. The Articulator and Hypothesis Partner guide executor agents (Domain-Specific Coding Agent and Code Evaluation Agent) to produce and assess Verilog designs rigorously. Our experiments demonstrate $\sim 25\%$ timing improvement and $\sim 22\%$ power reduction with failure rates $< 30\%$, with ablations showing all components and iterative refinement are essential for robust performance. Future directions include expanding the dataset, exploring adaptive iteration strategies, extending to additional RTL optimizations, leveraging retrieval-augmented prompting, full synthesis flow validation, and incorporating self-play or reinforcement learning to further enhance optimization outcomes.

Limitations

While our framework advances automated RTL optimization, several limitations remain. First, CODMAS has been evaluated primarily on pipelining and clock-gating transformations, and its generalization to broader categories of RTL optimizations (e.g., retiming, resource sharing, FSM restructuring) is not yet fully established. Second, although the evaluation pipeline incorporates deterministic EDA tools, scalability to very large industrial designs is constrained by tool runtime and the need for repeated synthesis queries. Third, the dialectic reasoning agents occasionally generate overly generic transformation plans that require iterative refinement, indicating that the system still relies on principled prompting and task-specific templates. Fourth, RTLOPT is a seed benchmark with limited size, and expanding it to capture the diversity of industrial RTL coding styles and multi-file hierarchies is an important direction for future work. Finally, because functional equivalence is verified using standard testbenches rather than exhaustive formal techniques, subtle corner-case divergences may go undetected in rare scenarios. These limitations highlight opportunities for improving reasoning robustness, dataset coverage, and scalability in future iterations of the framework.

Ethical Considerations

Although RTLOPT is built entirely from publicly licensed Verilog code, integrating it into an automated RTL-optimization workflow introduces certain security considerations. Prior work has demonstrated that LLM-generated RTL can contain vulnerabilities cataloged under Common Weakness Enumerations (CWEs) (Gadde et al., 2024). Furthermore, LLMs for HDL generation may be susceptible to data-poisoning or backdoor attacks, where compromised training data leads to the generation of insecure or malicious circuit components (Mankali et al., 2025). To mitigate these risks, our system emphasizes human oversight and interpretability by generating explicit transformation plans and hypotheses that expert designers can review and approve. We also perform rigorous simulation and synthesis checks to ensure deterministic validation and detect unintended structural or security flaws. All modules in RTLOPT are fully documented with origin and licensing information to support clear provenance tracking. Finally, we recommend that any deployment of automatically optimized hardware include additional security audits, formal verification, and human review, particularly in safety or security-critical applications.

References

AI@Meta. 2024. Llama 3 Model Card.

Mohammad Akyash, Kimia Azar, and Hadi Kamali. 2025. Rtl++: Graph-enhanced llm for rtl code generation. *arXiv preprint arXiv:2505.13479*.

Kaiyan Chang, Ying Wang, Haimeng Ren, Mengdi Wang, Shengwen Liang, Yinhe Han, Huawei Li, and Xiaowei Li. 2023. Chipgpt: How far are we from natural language hardware design. *arXiv preprint arXiv:2305.14019*.

Matthew DeLorenzo, Animesh Basak Chowdhury, Vasudev Gohil, Shailja Thakur, Ramesh Karri, Siddharth Garg, and Jeyavijayan Rajendran. 2024. Make every move count: Llm-based high-quality rtl code generation using mcts. *CoRR*.

Deepak Narayan Gadde, Aman Kumar, Thomas Nalapat, Evgenii Rezunov, and Fabio Cappellini. 2024. All artificial, less intelligence: Genai through the lens of formal verification. *arXiv preprint arXiv:2403.16750*.

Dario Garcia-Gasulla, Gokcen Kestor, Emanuele Parisi, Miquel Albert'i-Binimelis, Cristian Gutierrez, Razine Moundir Ghorab, Orlando Montenegro, Bernat Homs, and Miquel Moretó. 2025. Turtle: A unified evaluation of llms for rtl generation. *CoRR*.

Ce Guo and Tong Zhao. 2025. Resbench: A resource-aware benchmark for llm-generated fpga designs. In *Proceedings of the 15th International Symposium on Highly Efficient Accelerators and Reconfigurable Technologies*, pages 25–34.

Chia-Tung Ho, Haoxing Ren, and Brucek Khailany. 2024. VerilogCoder: Autonomous Verilog Coding Agents with Graph-based Planning and Abstract Syntax Tree (AST)-based Waveform Tracing Tool. *Preprint*, arXiv:2408.08927.

Yao Lai, Jinxin Liu, Zhentao Tang, Bin Wang, Jianye Hao, and Ping Luo. 2023. ChiPFormer: transferable chip placement via offline decision transformer. In *Proceedings of the 40th International Conference on Machine Learning (ICML)*, volume 202, pages 18346–18364. PMLR.

Aixin Liu, Bei Feng, Bing Xue, Bingxuan Wang, Bochao Wu, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenyu Zhang, Chong Ruan, and 1 others. 2024a. Deepseek-v3 technical report. *arXiv preprint arXiv:2412.19437*.

Mingjie Liu, Nathaniel Pinckney, Brucek Khailany, and Haoxing Ren. 2023. VerilogEval: Evaluating Large Language Models for Verilog Code Generation. In *2023 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE.

Shang Liu, Wenji Fang, Yao Lu, Qijun Zhang, Hongce Zhang, and Zhiyao Xie. 2024b. RTLCoder: Outperforming GPT-3.5 in Design RTL Generation with Our Open-Source Dataset and Lightweight Solution. *Preprint*, arXiv:2312.08617.

Yao Lu, Shang Liu, Qijun Zhang, and Zhiyao Xie. 2024. RTLLM: An Open-Source Benchmark for Design RTL Generation with Large Language Model. In *Proceedings of the 29th Asia and South Pacific Design Automation Conference (ASP-DAC)*, pages 722–727. IEEE.

Lakshmi Likhitha Mankali, Jitendra Bhandari, Manaar Alam, Ramesh Karri, Michail Maniatakos, Ozgur Sinanoglu, and Johann Knechtel. 2025. Rtl-breaker: Assessing the security of llms against backdoor attacks on hdl code generation. In *2025 Design, Automation & Test in Europe Conference (DATE)*, pages 1–7. IEEE.

Mayank Mishra, Matt Stallone, Gaoyuan Zhang, Yikang Shen, Aditya Prasad, Adriana Meza Soria, Michele Merler, Parameswaran Selvam, Saptha Surendran, Shivdeep Singh, Manish Sethi, Xuan-Hong Dang, Pengyuan Li, Kun-Lung Wu, Syed Zawad, Andrew Coleman, Matthew White, Mark Lewis, Raju Pavuluri, and 27 others. 2024. Granite Code Models: A Family of Open Foundation Models for Code Intelligence. *Preprint*, arXiv:2405.04324.

OpenAI. 2024. Chatgpt. `https://chatgpt.com/`.

Zehua Pei, Fangzhou Liu, Zhuolun He, Guojin Chen, Haisheng Zheng, Keren Zhu, and Bei Yu. 2023. AlphaSyn: Logic Synthesis Optimization with Efficient Monte Carlo Tree Search. In *2023 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE.

Nathaniel Pinckney, Chenhui Deng, Chia-Tung Ho, Yun-Da Tsai, Mingjie Liu, Wenfei Zhou, Brucek Khailany, and Haoxing Ren. 2025. Comprehensive verilog design problems: A next-generation benchmark dataset for evaluating large language models and agents on rtl design and verification. *arXiv preprint arXiv:2506.14074*.

Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Romain Sauvestre, Tal Remez, Jérémy Rapin, Artyom Kozhevnikov, Ivan Evtimov, Joanna Bitton, Manish Bhatt, Cristian Canton Ferrer, Aaron Grattafiori, Wenhan Xiong, Alexandre Défossez, and 7 others. 2024. Code Llama: Open Foundation Models for Code. *Preprint*, arXiv:2308.12950.

Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. 2024. Reflexion: language agents with verbal reinforcement learning. In *Proceedings of the 37th International Conference on Neural Information Processing Systems (NeurIPS)*. Curran Associates Inc.

Shailja Thakur, Baleegh Ahmad, Hammond Pearce, Benjamin Tan, Brendan Dolan-Gavitt, Ramesh Karri, and Siddharth Garg. 2024a. VeriGen: A Large Language Model for Verilog Code Generation. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 29(3):1–31.

Shailja Thakur, Jason Blocklove, Hammond Pearce, Benjamin Tan, Siddharth Garg, and Ramesh Karri. 2024b. AutoChip: Automating HDL Generation Using LLM Feedback. *Preprint*, arXiv:2311.04887.

Kiran Thorat, Jiahui Zhao, Yaotian Liu, Hongwu Peng, Xi Xie, Bin Lei, Jeff Zhang, and Caiwen Ding. 2024. Advanced Large Language Model (LLM)-Driven Verilog Development: Enhancing Power, Performance, and Area Optimization in Code Synthesis. *Preprint*, arXiv:2312.01022.

Prashanth Vijayaraghavan, Apoorva Nitsure, Charles Mackin, Luyao Shi, Stefano Ambrogio, Arvind Haran, Viresh Paruthi, Ali Elzein, Dan Coops, David Beymer, and 1 others. 2024. Chain-of-descriptions: Improving code llms for vhdl code generation and summarization. In *Proceedings of the 2024 ACM/IEEE International Symposium on Machine Learning for CAD*, pages 1–10.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. 2023. React: Synergizing reasoning and acting in language models. In *International Conference on Learning Representations (ICLR)*.

Xufeng Yao, Yiwen Wang, Xing Li, Yingzhao Lian, Ran Chen, Lei Chen, Mingxuan Yuan, Hong Xu, and Bei Yu. 2024. RTLRewriter: Methodologies for Large Models aided RTL Code Optimization. In *2024 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE.

Zhongzhi Yu, Mingjie Liu, Michael Zimmer, Yingyan Celine, Yong Liu, and Haoxing Ren. 2025. Spec2rtl-agent: Automated hardware code generation from complex specifications using llm agent systems. In *2025 IEEE International Conference on LLM-Aided Design (ICLAD)*, pages 37–43. IEEE.

Qihao Zhu, Daya Guo, Zhihong Shao, Dejian Yang, Peiyi Wang, Runxin Xu, Y Wu, Yukun Li, Huazuo Gao, Shirong Ma, and 1 others. 2024. DeepSeek-Coder-V2: Breaking the Barrier of Closed-Source Models in Code Intelligence.

A Algorithm: CODMAS

Algorithm 1 CODMAS Optimization Loop

Require: Input RTL C_0, Opt. Goal G, Iteration Cap T

$P \leftarrow \texttt{ArticulatorInit}(C_0, G)$
$H \leftarrow \texttt{HypoPartnerInit}(C_0, G)$
$D \leftarrow \texttt{DataflowGraph}(C_0)$
$\Pi \leftarrow \texttt{DKI}(P, H, D)$
$C \leftarrow \texttt{LLMGenerate}(C_0, \Pi)$
$(E_{\text{syn}}, E_{\text{func}}, M_{\text{ppa}}) \leftarrow \texttt{CEA}(C, C_0)$
$t \leftarrow 0$
while $(E_{\text{syn}} \neq \emptyset$ **or** $E_{\text{func}} \neq \emptyset$ **or** $M_{\text{ppa}} \not\geq G)$ **and** $t < T$ **do**
  if $E_{\text{syn}} \neq \emptyset$ **then**
    $E_{\text{func}} \leftarrow \emptyset$, $M_{\text{ppa}} \leftarrow \emptyset$
    $P \leftarrow \texttt{ArticulatorUpdate}(P, E_{\text{syn}})$
  else if $E_{\text{func}} \neq \emptyset$ **then**
    $M_{\text{ppa}} \leftarrow \emptyset$
    $H \leftarrow \texttt{HypoPartnerUpdate}(H, E_{\text{func}})$
    $P \leftarrow \texttt{ArticulatorAssist}(P, E_{\text{func}})$
  else if $M_{\text{ppa}} \not\geq G$ **then**
    $P \leftarrow \texttt{ArticulatorUpdate}(P, M_{\text{ppa}})$
    $H \leftarrow \texttt{HypoPartnerUpdate}(H, M_{\text{ppa}})$
  end if
  $\Pi \leftarrow \texttt{DKI}(P, H, D)$
  $C \leftarrow \texttt{LLMGenerate}(C_0, \Pi)$
  $(E_{\text{syn}}, E_{\text{func}}, M_{\text{ppa}}) \leftarrow \texttt{CEA}(C, C_0)$
  $t \leftarrow t + 1$
end while
return Final optimized RTL C

B Metric Computation Details

Optimized RTL designs are evaluated along four key dimensions: Area (A), Power (P), Timing (T), and Failure Rate (FR). All metrics are computed using Yosys with Liberty-based standard-cell libraries.

Area (A): Computed as the total synthesized cell area. Reported relative to baseline as A/A_0, where A_0 is the unoptimized design. Values < 1 indicate area reduction, while > 1 indicate an increase.

Power (P): Estimated static and dynamic power consumption using Yosys synthesis reports. Percentage improvement is computed as

$$P_{\%} = \frac{P_0 - P}{P_0} \times 100$$

where P_0 is the power of the original RTL.

Timing (T): Measured by critical path delay (CPD), defined as the longest combinational path delay through library cells. Percentage improvement is

$$T_{\%} = \frac{CPD_0 - CPD}{CPD_0} \times 100$$

where CPD_0 is the baseline.

Failure Rate (FR): Fraction of RTL designs that fail verification or synthesis checks after optimization. A design is considered failed if it violates functional correctness, fails synthesis, or exceeds timing constraints. Lower FR indicates higher reliability and robustness of the optimization framework. All metrics are reported per design, with averages and standard deviations provided across the dataset for aggregate evaluation. CPD is adjusted for structural transformations such as pipelining (accounting for latency shifts) to ensure fair comparison. Power and area are normalized by baseline RTL to facilitate cross-design comparison.

C Datasets

To evaluate automated RTL optimization, we introduce RTLOPT, a dataset designed for reproducible and metric-driven assessment. Table 4 compares RTLOPT against prior Verilog datasets. Existing benchmarks such as VerilogEval (Liu et al., 2023) and TuRTLe (Garcia-Gasulla et al., 2025) focus on functional correctness but lack synthesizability or metric-specific evaluation, limiting their utility for evaluating PPA-aware transformations. Other datasets, including RTLRewriter (Yao et al., 2024) and ResBench (Guo and Zhao, 2025), provide synthesizable RTL but do not include functional tests or metric-oriented targets, restricting systematic assessment of optimization performance.

RTLOPT contains 120 Verilog triples (unoptimized, optimized, and testbench), all synthesizable and functionally validated, with clearly defined PPA objectives such as pipelining and clock gating. This allows rigorous evaluation of both correctness and optimization effectiveness, enabling quantitative comparisons across LLM-driven frameworks. By explicitly including metric-specific targets, RTLOPT fills a critical gap, providing a standardized benchmark for evaluating end-to-end RTL optimization pipelines in a reproducible manner.

Dataset	Size	Functionality	Synthesizability	Metric-specific
VerilogEval (Liu et al., 2023)	156	✓	✗	✗
RTLLM (Lu et al., 2024)	30	✓	✓	✗
RTLRewriter (Yao et al., 2024)	95	✗	✓	✗
TuRTLe (Garcia-Gasulla et al., 2025)	223	✓	✓	✗
CVDP (Pinckney et al., 2025)	783	✓	✓	✗
ResBench (Guo and Zhao, 2025)	56	✓	✓	✗
RTLOPT (Ours)	120	✓	✓	✓

Table 4: Comparison of RTLOPT with prior Verilog datasets. "Metric-specific" indicates whether metric-specific optimization exists or not for evaluation.

Module	Original RTL	CODMAS Optimized RTL
Multiplier	Sequential multiply w/o pipeline	Partial product pipelined with inter-stage registers
Adder	Ripple-carry 32-bit adder	Pipelined adder with reduced critical path
Control Logic	Unconditioned enable signals	Handshake-aware control signals with proper gating
Observed Errors in Baselines	Syntax errors, functional mismatches, unmet PPA targets	
Corrected by CODMAS	All syntax errors fixed, functional simulation passes, critical path reduced $\sim 25\%$	

Table 5: Example of pipelining transformation and error mitigation under CODMAS. Dialectic agents guide structured edits, and the CEA validates correctness and PPA improvements.

CODMAS Performance: Mean $\pm$ Std over 5 runs						
Model	**Pipelining**			**Clock Gating**		
	A (↓)	T (↑)	FR (↓)	A (↓)	P (↑)	FR (↓)
GPT-4o	0.960 ± 0.007	25.5 ± 1.2	19.5 ± 2.1	0.999 ± 0.006	21.7 ± 1.5	21.8 ± 2.3
GPT-3.5-turbo	0.972 ± 0.009	21.3 ± 1.1	23.4 ± 2.4	1.020 ± 0.007	18.8 ± 1.2	24.2 ± 2.0
DeepSeek-v2.5	0.979 ± 0.010	21.4 ± 1.0	22.8 ± 2.3	1.015 ± 0.008	19.0 ± 1.3	23.6 ± 2.1
Llama-3	1.032 ± 0.012	19.8 ± 1.3	25.5 ± 2.5	1.048 ± 0.010	16.5 ± 1.1	26.3 ± 2.4
Granite-34b	0.998 ± 0.011	10.5 ± 0.9	28.3 ± 2.8	1.030 ± 0.009	12.9 ± 1.0	29.5 ± 2.6
CodeLlama-34b	1.030 ± 0.012	11.2 ± 1.0	29.5 ± 2.7	1.042 ± 0.010	10.6 ± 0.9	31.3 ± 2.8

Table 6: Extracted CODMAS results with mean and standard deviation over five runs. Metrics: area (A), timing (T), power (P), and failure rate (FR). FR denotes fraction of runs failing syntax, functional, or PPA checks.

D Baseline Details

D.1 Baseline Methods

Zero-Shot Prompting. Models receive a single instruction describing optimization goals (area, timing, power) and are asked to produce an optimized Verilog implementation in one shot. No iterative reasoning, feedback, or correction is provided. This baseline captures the lower bound of LLM-only optimization.

CoDes (Chain-of-Descriptions). Following (Vijayaraghavan et al., 2024), the model generates a sequence of descriptive intermediate transformations—structural changes, expected effects on PPA, and planned optimizations—before emitting code. We adapt CoDes to explicitly reference RTL constructs, combinational paths, and pipeline boundaries.

ReAct. ReAct (Yao et al., 2023) interleaves reasoning traces with "actions." For RTL, actions correspond to producing partial code, checking syntax, or querying simulation outputs. The model reasons about identified issues and attempts corrections but lacks the deeper structural planning used in CODMAS.

Reflexion. Reflexion (Shinn et al., 2024) enables the model to store brief textual "reflections" describing causes of failures (e.g., functional mismatch or timing regression). Reflections form an episodic memory across attempts. For RTL tasks, reflections include mis-structured pipeline stages, incorrect sensitivity lists, or inferred latches.

LLM-VeriPPA. LLM-VeriPPA (Thorat et al., 2024) uses a two-stage process: (1) correct syntax and functional behavior, (2) re-prompt the model to improve PPA while preserving equivalence. We

reproduce this pipeline and ensure that verification checks match those used for CODMAS, including testbench simulation and Yosys-based PPA extraction.

D.2 Model Backends

All baselines are evaluated under identical compilation, simulation, and Yosys Liberty-based synthesis flows. We test proprietary models (GPT-4o, GPT-3.5-turbo) (OpenAI, 2024) and open-source models (Llama-3 (AI@Meta, 2024), DeepSeek-v2.5 (Zhu et al., 2024), Granite-34B-Code (Mishra et al., 2024), CodeLLaMA-34B (Rozière et al., 2024)). Temperature, sampling parameters, and code-length limits follow standard practice and are documented for reproducibility. Each baseline originally targets generic code generation; we adapt them for RTL-specific optimization by: (1) enforcing functional equivalence via testbench simulation, (2) integrating PPA feedback loops where relevant, and (3) constraining all methods to the same maximum number of attempts and evaluation budget. These details ensure that comparisons isolate algorithmic differences rather than evaluation infrastructure.

D.3 Implementation Details and Qualitative Analysis

All experiments were conducted on a server with 64-core CPU and NVIDIA A100 GPUs. We evaluate CODMAS on six LLMs using the RTLOPT benchmark. Each experiment is repeated five times with different random seeds to account for stochastic variation. Metrics include area (A), timing (T) for pipelining, power (P) for clock gating, and failure rate (FR). Reported results correspond to mean $\pm$ standard deviation across runs. Paired two-sample t-tests confirm that improvements over baselines are statistically significant ($p < 0.01$).

Table 5 illustrates a representative pipelining transformation. Baseline RTL often exhibits syntax errors, functional mismatches, and unmet PPA targets, with typical failure distributions of $\sim 40\%$ syntax, $\sim 35\%$ functional, and $\sim 25\%$ PPA violations. In this example, sequential multipliers and ripple-carry adders create long critical paths, while control logic lacks proper gating. CODMAS applies structured edits guided by the dialectic agents: pipelining arithmetic units with inter-stage registers, optimizing critical paths, and enforcing handshake-aware control signals. The Code Evaluation Agent (CEA) validates syntax, functional correctness, and PPA improvements. Across five runs, the optimized RTL achieves $\sim 25.5\% \pm 0.6$ reduction in critical path delay, $\sim 21.7\% \pm 0.5$ power reduction under clock gating, and failure rates consistently below 30%, demonstrating robustness and reproducibility. This example highlights how the reasoning–evaluation loop systematically corrects errors while improving performance, particularly for smaller or otherwise weaker models.

Error analysis indicates that baseline failures arise from a mix of syntax errors ($\sim 40\%$), functional mismatches ($\sim 35\%$), and unmet PPA objectives ($\sim 25\%$). CODMAS mitigates all three categories through iterative reasoning-guided refinement, combining the Articulator's plan, the Hypothesis Partner's predictions, and deterministic evaluation by the CEA.