LEGO: An LLM Skill-Based Front-End Design Generation Platform

Jincheng Lou[1], Ruohan Xu[2], Jiecheng Ma[3], Runzhe Tao[1], Xinyu Qu[1], and Yibo Lin[1,4,5,*]
[1]School of IC, Peking University, [2]School of EECS, Peking University,
[3]School of Microelectronics, Xidian University, [4]Institute of EDA, Peking University,
[5]Beijing Advanced Innovation Center for IC
Email: jinchenglou@stu.pku.edu.cn [*]Corresponding author: yibolin@pku.edu.cn

***Abstract*—Existing LLM based EDA agents are often isolated task-specific systems. This leads to repeated engineering effort and limited reuse of successful design and debugging strategies. We present LEGO, a unified skill-based platform for front-end design generation. It decomposes the digital front-end flow into six independent steps and represents every agent capability as a standardized composable circuit skill within a plug-and-play architecture. To build this skill library, we survey more than 100 papers, select 11 representative open-source projects, and extract 42 executable circuit skills within a six-step finite state machine formulation. Circuit Skill Builder automates skill extraction with linear scalability. Agent Skill RAG achieves submillisecond retrieval without relying on embedding models. Empirical evaluation on a hard subset of 41 VerilogEval v2 problems that gpt-5.2-codex fails to solve under extra-high (xhigh) reasoning effort shows that individual circuit skills constructed within LEGO raise Pass@1 from 0.000 to 0.805. This is an 80.5% gain over the baseline. Cross project skill compositions also reach 0.805 Pass@1. They outperform `hierarchy-verilog` by 14.6% and `VerilogCoder` by 2.5%. They also match `MAGE`. These results show that modular skill composition supports both effective and flexible RTL design automation. The LEGO platform and all circuit skills are publicly available at GitHub (`github.com/loujc/LEGO-An-LLM-...`).**

***Index Terms*—LLM Agents, Agent Skills, RTL Generation, Front-End Design Flow, VerilogEval**

I. INTRODUCTION

Automating the digital front-end design flow is a central challenge in AI assisted chip design. The flow covers specification refinement, RTL coding, testbench generation, simulation, and iterative debugging. Generated RTL code must satisfy strict syntax rules, timing constraints, and signal-level functional correctness. This breadth and the stringent requirements at each step make end-to-end automation difficult and expensive. Large language models and code agents have shown strong potential in this domain. Prior work has made progress in benchmark construction [1]–[5], agent-based design workflows [6]–[10], and domain-specific LLM training [11]–[16]. Most existing systems are isolated solutions. They bind algorithmic ideas to project-specific toolchains. This limits cross-project reuse and forces repeated implementation of common infrastructure.

To address this fragmentation, we propose LEGO, a unified skill-based front-end design generation platform. Fine-tuning approaches and multi-agent systems often reimplement infrastructure for partial flows and incur high LLM API cost. LEGO instead decomposes the workflow into six-steps and supports plug-and-play composition across steps. As shown in Fig. 1, LEGO runs on top of code agent tools. It can therefore use vendor provided coding plans. These plans are more affordable and accessible than raw LLM API calls. Circuit Skill Builder converts open-source projects into LEGO compliant circuit skills within the open code agent ecosystem. Agent Skill RAG accumulates design and debugging experience for efficient reuse.

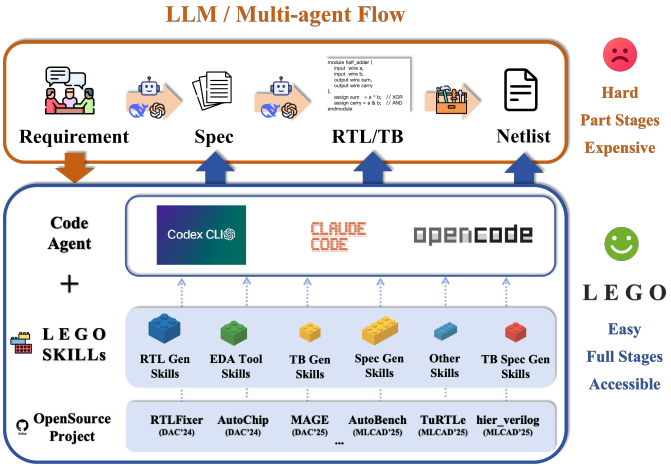

Fig. 1. LEGO System Overview

Our contributions are listed below.

- **LEGO is the first open skill-based platform for digital front-end design.** It decomposes the workflow into six independent steps and supports plug-and-play composition. This enables cross-project reuse without rebuilding infrastructure. The platform and extracted circuit skills are publicly released in a unified GitHub repository.
- **Executable skill library.** We curate a library of 42 executable circuit skills extracted from 11 representative open-source projects and organized into 24 functional groups. Circuit Skill Builder automates skill extraction. Agent Skill RAG supports lightweight experience accumulation and achieves submillisecond retrieval.
- **Strong empirical gains.** We validate the framework on a hard subset of 41 VerilogEval v2 problems from tasks that

gpt-5.2-codex fails to solve. The results show an 80.5% Pass@1 improvement over the baseline. Composed skills outperform hierarchy-verilog by 14.6% and VerilogCoder by 2.5%. They also match the RTL generation SOTA method MAGE.

II. Preliminary

A. LLM and Multi-Agent Paradigms in RTL Generation

Compared with software languages, Verilog and other HDLs encode hardware hierarchy, timing behavior, and signal dependencies in a tightly coupled form. RTL generation therefore requires coordinated reasoning across specification understanding, code synthesis, simulation feedback, and iterative correction, making the task difficult for a single static prompting strategy.

Current approaches to LLM assisted RTL generation follow two main directions. The first direction trains domain-specific LLMs on hardware description data [11]–[16]. However, the scale of publicly available RTL datasets is orders of magnitude smaller than that of software code. The structural gap between HDL and general-purpose programming languages further limits transfer. These factors make domain-specific models expensive to train and their performance still lags behind expectations. The second direction leverages multi-agent systems [6]–[10]. These systems benefit directly from the rapid improvement of general-purpose LLMs and have become the mainstream approach. Multi-agent systems decompose the workflow into specialized roles with step-specific tools and prompts. This decomposition reduces context interference across heterogeneous subtasks such as specification analysis, RTL coding, testbench generation, and debug. Our framework follows this collaborative principle and further standardizes each executable capability as a reusable skill unit.

B. Code Agents and the Abstraction of Skills

A code agent is an autonomous programming system that can invoke external tools and react to execution feedback. Representative code agents include Codex CLI [17], Claude Code [18], and OpenCode [19]. Beyond token generation, these agents execute shell commands, interpret compiler and simulator outputs, and perform iterative error correction. The digital front-end design flow is predominantly composed of command-line invocations, structured text processing, and code manipulation, making code agents a natural fit for this domain.

These code agent platforms provide an official abstraction called Skill as an atomic capability unit for environment interaction and structured reasoning. A skill encapsulates a single well-defined capability such as generating RTL code, running simulation, or extracting error messages. It acts as a self-contained module with clearly defined inputs, outputs, and execution logic. This enables composition, reuse, and independent updates. Consistent with the seven field skill specification in Section III-B, the formal representation of a skill is

$$S = (N, F, C, P, IO, \Sigma, G) \tag{1}$$

Algorithm 1 Circuit Skill Builder

Input Project P, paper R, and taxonomy $\mathcal{T} = \{w_1, \ldots, w_6\}$
Output Skill set $\mathcal{S}_{\text{new}}$ and skill groups $\{\mathcal{G}_j\}$

Step 1. Summarize // Extract capabilities
$\mathcal{M} \leftarrow$ Summarize innovations and functions from P and R using LLM

Step 2. Extract // Map to workflow steps
$\mathcal{S}_{\text{cand}} \leftarrow \emptyset$
for each capability $m \in \mathcal{M}$ **do**
  Identify target step $w_i \in \mathcal{T}$ for m
  Generate $(N, F, C, P, IO) \leftarrow$ Extract fields from m and P codebase
  $\mathcal{S}_{\text{cand}} \leftarrow \mathcal{S}_{\text{cand}} \cup \{(N, F, C, P, IO)\}$
end for

Step 3. Post process // Standardize and organize
for each $s \in \mathcal{S}_{\text{cand}}$ **do**
  Complete s with schema Σ and done criteria G
end for
$\mathcal{S}_{\text{new}} \leftarrow \text{Deduplicate}(\mathcal{S}_{\text{cand}})$
Classify $\mathcal{S}_{\text{new}}$ into $\mathcal{S}_1, \ldots, \mathcal{S}_6$ by workflow step
Compose groups $\{\mathcal{G}_j\}$ based on functional complementarity

return $\mathcal{S}_{\text{new}}$, $\{\mathcal{G}_j\}$

where N is the skill name, F is the simple function description, C is the constraints section, P is the entrypoint command, IO is the input and output specification, Σ is the schema definition, and G is the done criteria. This abstraction provides a consistent interface for composition, execution, and experience accumulation.

III. Methodology

A. Workflow Decomposition and Skill Curation

As shown in Fig. 1, LEGO decomposes the digital front-end design flow into six sequential steps. These steps are RTL Spec Generation, Testbench Spec Generation, RTL Generation, Testbench Generation, EDA Tool, and Others. They cover specification refinement, code synthesis, compilation, simulation with tools such as Icarus Verilog [20] and Verilator [21], and auxiliary utilities.

We model this workflow as a finite state machine. Let $\mathcal{W} = \{w_1, \ldots, w_6\}$ denote the workflow steps, and let $\mathcal{S}_i$ denote the skill set at step w_i. The state at iteration t is

$$\Psi^{(t)} = \left(w_i, s_j^{(i)}, \mathcal{A}^{(t)}, \mathcal{D}^{(t)}\right) \tag{2}$$

where $w_i \in \mathcal{W}$ is the current step, $s_j^{(i)} \in \mathcal{S}_i$ is the active skill, $\mathcal{A}^{(t)}$ are accumulated artifacts, and $\mathcal{D}^{(t)}$ tracks design decisions. The transition function δ maps $\Psi^{(t)}$ to $\Psi^{(t+1)}$ and governs step progression based on EDA tool feedback.

TABLE I
CIRCUIT SKILL LIST AND GROUP NOTATIONS IN LEGO

Step Skills	Group	#	Included Circuit Skills
RTL Spec	S1	8	hier_submodule_list_prompt, hier_outline_prompt, hier_subhier_json_prompt, hier_function_check_prompt, hier_header_check_prompt, hier_refine_question_prompt, hier_refine_integrate_prompt, prompt_enricher
	S2	1	spec2rtl-understanding-pipeline
TB Spec	TS1	1	autobench_circuit_type_classify
	TS2	1	autobench_tb_scenarios_prompt
	TS3	1	autobench_tb_rules_extract
	TS4	1	autobench_tb_spec_prompt
	TS5	1	iverilog_waveform_parser
	TS6	1	mage_sim_judge_tb_fix
	TS7	1	verilogcoder_case_loader
RTL Gen	G1	1	verilogcoder_rtl_subtask_prompt
	G2	1	mage_rtl_generate
	G3	1	autobench_rtl_prompt
	G4	2	hier_verilog_gen_prompt, spec_ir_codex_rtl_gen
TB Gen	TG1	1	autobench_directgen_prompt
	TG2	1	autobench_pychecker_tb_prompt
	TG3	1	verilogcoder_tb_merge
EDA Tool	E1	3	iverilog_compile, iverilog_syntax_fixer, spec2rtl-closed-loop
	E2	5	iverilog_compile, iverilog_error_localizer, iverilog_error_rag, iverilog_code_fixer, iverilog_compile_fix_chain
	E3	3	spec2rtl-closed-loop waveform_feedback, iverilog_simulator, verilogcoder_waveform_trace
Others	O1	2	iverilog_error_localizer, iverilog_error_rag
	O2	1	autobench_runinfo_analyze
	O3	2	mage_token_accounting, mage_benchmark_reader
	O4	1	rtl_fault_injector
	O5	1	hier_tree_ops
Total: 6 stages, 24 skill groups, 42 circuit skills			

To construct the skill sets $\mathcal{S}_i$, we survey over 100 papers on LLM assisted front-end design and select 11 representative open-source projects [8]–[10], [22]–[29]. From these projects, 42 standardized circuit skills are extracted and organized into $\mathcal{S}_1$ through $\mathcal{S}_6$ across all six-steps as shown in Table I. This systematic formulation, combined with skill construction from proven sources, enables up to 70.7% Pass@1 improvement over the baseline in Experiments 1 and 2.

B. Skill Specification and Construction

To ensure cross-project compatibility, each circuit skill follows the seven field specification in Eq. (1). The fields are name N, function description F, constraints C, entrypoint P, input and output specification IO, schema Σ, and done criteria G. The naming pattern `<project_abbr>_<function_abbr>` guarantees uniqueness, while the function description enables the agent to select the appropriate skill at runtime. The entrypoint P specifies the command-line invocation, IO and Σ define data structures and formats, and G determines completion conditions.

Circuit Skill Builder in Algorithm 1 automates skill construction from open-source projects. Applying this pipeline to 11 projects yields 42 circuit skills organized into 24 groups in Table I. Skills are grouped by functional complementarity within each workflow step, ensuring similar input/output interfaces for interchangeable composition. For instance, RTL Gen groups G1 to G4 represent distinct methodologies such as subtask decomposition, direct generation, specification-driven generation, and hierarchical generation. They come from different source projects. Beyond these circuit skills, LEGO provides Agent Skill RAG to accumulate debugging and design experience for each workflow step. Unlike traditional vector-based RAG, Agent Skill RAG employs a lightweight two-stage retrieval mechanism. First, the agent loads only the concise descriptions of all available RAG entries in each skill, maintaining minimal context overhead. When a relevant entry is identified through semantic matching, the agent then loads the complete knowledge unit with detailed fields such as symptom pattern, root cause, fix strategy, and applicable conditions. This lazy loading approach eliminates the need for embedding models or reranking while enabling submillisecond initial retrieval and real-time knowledge updates via simple text appends. Retrieved fixes are injected as explicit prompts to guide subsequent iterations.

C. Hierarchical Skill Architecture

As shown in Fig. 2, the skill system of LEGO comprises two components. They are Template and Step Skill. A Template encapsulates a preset combination of circuit skills, enabling users to initiate design tasks directly without manual configuration. Within a Template, users can also specify which circuit

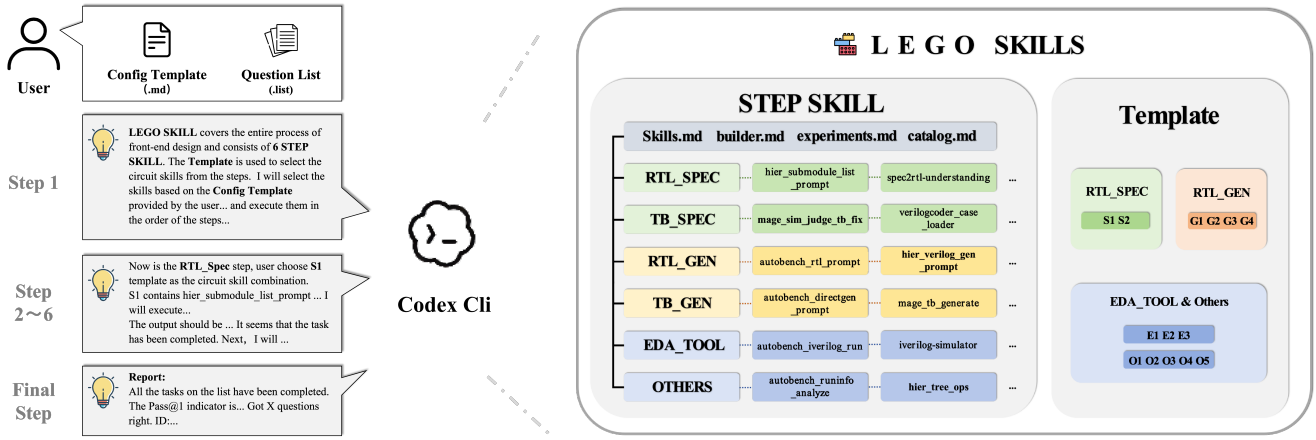

Fig. 2. Step level view of the LEGO methodology.

skills to activate for each step, thereby adapting the system to specific requirements.

The Step Skill component forms a three-layer hierarchy together with the top level LEGO Skill. The top layer defines the overall workflow decomposition, execution order, and iteration logic. The middle layer consists of six-step skills, each listing available circuit skills and specifying active configurations dynamically adjusted via the config template. The bottom layer contains the circuit skills and Agent Skill RAG entries with their complete definitions, forming the atomic execution units of LEGO.

The entire system is lightweight and highly extensible because it consists primarily of markdown files and requires no complex environment setup. As illustrated in the right panel of Fig. 2, the execution workflow is straightforward. The user provides a config template and question list. The code agent invokes the LEGO Skill and loads the configuration to select active circuit skills for each step. It then proceeds through the six-steps in order while preserving intermediate results. Finally it generates a summary report upon completion.

IV. EXPERIMENTS

This section presents three complementary analyses. Subsection IV-A describes the experimental setup, including the model, evaluation metric, and benchmark construction. Subsection IV-B evaluates the effectiveness of step decomposition and individual circuit skills. Subsection IV-C studies composed skill pipelines and compares different combinations to validate the flexibility and effectiveness of LEGO.

A. Experimental Setup

To ensure that the behavior of each constructed circuit skill remains as close as possible to its source project, we use the official repositories reported in the corresponding papers for all open-source projects except VerilogAssistant. For VerilogAssistant we use an open-source reproduction repository on GitHub [30]. To evaluate the generality and effectiveness of the constructed skills, we do not install LEGO skills into the Codex directory, which allows us to test whether the model can autonomously use the constructed skills. All experiments use Codex CLI version 0.98.0 as the code agent platform with gpt-5.2-codex and extra-high (xhigh) reasoning effort. Every interaction with the LLM is performed directly through Codex CLI rather than API calls.

Our benchmark uses the specification-to-RTL tasks from VerilogEval v2 [2]. Unlike VerilogEval v1 [1], VerilogEval v2 includes only HumanEval style problems and emphasizes one-shot success rate Pass@1. This better aligns with the single generation setting in practical engineering. To better match practical engineering scenarios and more clearly reveal the performance gains achieved by LEGO skills, we filter the dataset. VerilogEval v2 contains 156 specification-to-RTL tasks, and a large portion of them can be solved directly by modern LLMs. We run end-to-end generation on VerilogEval v2 with gpt-5.2-codex and select the 41 difficult tasks that gpt-5.2-codex fails to solve as the benchmark for this work. Following prior work [8]–[10], [28], we compute Pass@1 as

$$\text{pass@}k = \mathbb{E}_{\text{Problems}}\left[1 - \frac{\binom{n-c_p}{k}}{\binom{n}{k}}\right] \tag{3}$$

where $k = 1$ and c_p is the number of passing runs. This Pass@1 metric accounts for multiple runs and reflects the expected percentage of problems that the system solves correctly when executed once for each problem. To evaluate skill composition flexibility, Experiment 2 constructs cross-project combinations that cover diverse design methodologies such as hierarchical methods, specification-driven methods, and understanding-based methods. It also covers complete workflow stages such as specification generation, RTL generation, and debugging. This ensures fair comparison with baseline works.

Tables II and III report solved-problem counts and Pass@1 for Experiments 1 and 2, respectively. Figure 3 visualizes correctness for each problem across the same settings in a grouped layout, with problem IDs on the horizontal axis explicitly showing every problem ID because the benchmark IDs are not consecutive. For settings with iterative execution, the

TABLE II
AGGREGATE METRICS OF EXPERIMENT 1 ON 41 FILTERED VERILOGEVAL V2 PROBLEMS.

Metric	gpt-5.2-codex (xhigh)	G1	G2	S1G1	G1E1E2 (w. loop)	G1E3 (w. loop)
Solved	0 / 41	23 / 41	25 / 41	22 / 41	29 / 41	**33 / 41**
Pass@1	0.000	0.561	0.610	0.537	0.707	**0.805**
Gain	+0.0%	+56.1%	+61.0%	+53.7%	+70.7%	**+80.5%**

Note. Gain is computed relative to gpt-5.2-codex under xhigh. Notation such as G1, S1G1, and E1 follows Table I.

TABLE III
AGGREGATE METRICS OF EXPERIMENT 2 ON 41 FILTERED VERILOGEVAL V2 PROBLEMS.

Metric	S1G4	S1G4E1E2 (w. loop)	S1G4E3 (w. loop)	S2E1E2E3 (w. loop)	hierarchy-verilog [24]	verilogcoder [8]	MAGE [10]	RTLFixer [28]
Solved	21 / 41	30 / 41	**33 / 41**	**33 / 41**	27 / 41	32 / 41	**33 / 41**	17 / 41
Pass@1	0.512	0.732	**0.805**	**0.805**	0.659	0.780	**0.805**	0.415
Gain	+51.2%	+73.2%	**+80.5%**	**+80.5%**	+65.9%	+78.0%	**+80.5%**	+41.5%

Note. Gain is computed relative to gpt-5.2-codex under xhigh. Notation follows Table I.

scheme label includes a second line "with loop", and overlaid numbers indicate nonzero loop counts for the corresponding problems.

In Experiment 2, the row and column corresponding to the prior work comparison method hierarchy-verilog are highlighted with a dedicated color to separate external comparison targets from compositions built in LEGO.

B. Workflow Decomposition and Skill Construction

To verify the effectiveness of the step decomposition and circuit skill construction, Experiment 1 adds one step-level skill group at a time and compares each setting with the baseline gpt-5.2-codex under xhigh. Because the 41 task benchmark is selected from problems that the baseline fails in a previous end-to-end run, the baseline accuracy is zero. As reported in Table II, adding G1 and G2 as RTL Generation skills improves Pass@1 by 56.1% and 61.0% over the baseline.

We then add S1 from the RTL Spec step on top of the RTL Generation setting to form S1G1. This setting solves one fewer problem than G1. A likely reason is that most tasks in VerilogEval v2 are single module problems with limited need for complex specification decomposition, while the RTL Generation skills already perform lightweight prompt adaptation. Finally, we add the loop-enabled settings E1E2 and E3. They yield clear gains of 70.7% and 80.5% over the baseline. The performance improvement from E2 and E3 is largely attributed to Agent Skill RAG, which provides structured debugging experience during the ReAct loop. Figure 3 shows that the overlaid loop counts are generally small. This indicates that most problems need only a few iterations to reach correct solutions. The solved problems are distributed across the entire problem ID range instead of concentrating in the lower IDs. This result shows that the constructed circuit skills, the decoupled workflow steps, and the ReAct loop mechanism are effective for solving problems of varying difficulty.

C. Cross Project Skill Composition Comparative Evaluation

To validate the flexibility and effectiveness of modular skill composition, we construct representative combinations that correspond to recognizable design methodologies from baseline works. S1G4 combines the specification decomposition of hierarchy-verilog, denoted as S1, with hierarchical generation, denoted as G4. It represents the complete hierarchy-verilog workflow [24]. S1G4E1E2 and S1G4E3 augment this baseline with debugging capabilities from spec2rtl [9] and RTLFixer [28], testing whether advanced iterative repair can enhance hierarchical methods. S2E1E2E3 starts from the understanding pipeline of spec2rtl, denoted as S2, and adds comprehensive debugging through E1E2E3. This setting represents an understanding first methodology. These combinations ensure functional complementarity across workflow stages while enabling direct comparison with their source projects.

We use S1G4 as the internal baseline for measuring composition gains and include hierarchy-verilog [24], VerilogCoder [8], and MAGE [10] as external prior work comparisons. The S1G4 baseline already improves Pass@1 by 51.2% over gpt-5.2-codex under xhigh. We then evaluate multiple cross-project skill compositions. As shown in Table III, S1G4E1E2 and S1G4E3 add loop-enabled combinations from spec2rtl [9] and RTLFixer [28] on top of S1G4. They improve by 22.0% and 29.3% over S1G4. Another high performing cross source combination is S2E1E2E3. It combines the understanding pipeline of spec2rtl with comprehensive debugging capabilities and improves by 29.3% over S1G4.

Comparing the composed skills with the original works, S1G4E1E2 outperforms hierarchy-verilog [24] by 7.3%. S1G4E3 and S2E1E2E3 both outperform hierarchy-verilog by 14.6% and VerilogCoder [8] by 2.5%. They also match MAGE [10]. In addition, they exceed standalone RTLFixer [28] by 39.0%. In isolation, RTLFixer is constrained by a single tool local repair loop and incomplete task context. It often produces syntactic patches that violate behavioral constraints. When integrated into LEGO, upstream decompo-

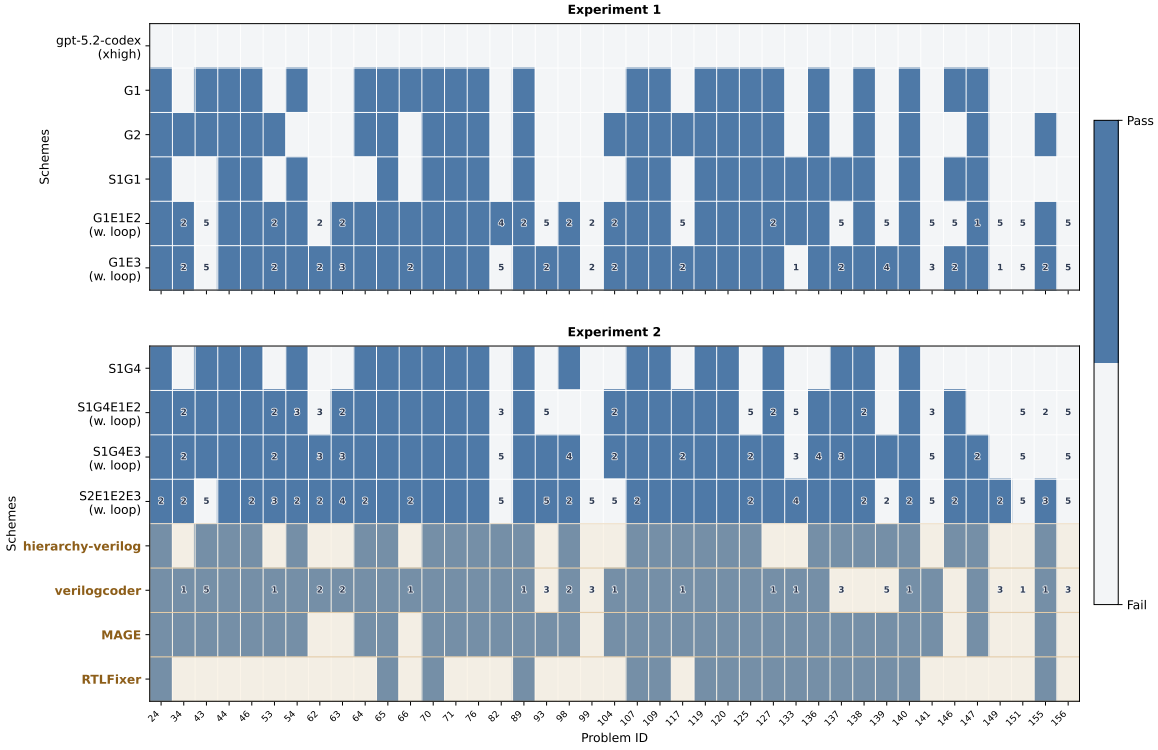

Fig. 3. Per problem results of Experiments 1 and 2 on 41 filtered VerilogEval v2 problems. The heatmap is split into two aligned subplots for Experiment 1 and Experiment 2. Binary correctness is shown for all settings with dark blue for pass and light gray for fail. All problem IDs are explicitly shown on the horizontal axis. Settings with iterative execution are marked with "with loop" on the second line of the scheme label, and overlaid numbers indicate nonzero loop counts.

sition and retrieval supply targeted guidance, while cross skill constraint checking prunes invalid fixes. This turns open-ended repair into guided correction. The multiskill feedback loop includes failure attribution, retry scheduling, and patch validation. It improves stability and explains the large empirical gain.

Figure 3 shows that each composed setting corrects a different subset of previously failed problems, further confirming the complementary nature of skills from different sources.

V. Conclusion

This paper introduces LEGO, a unified skill-based platform for digital front-end design generation. It decomposes the workflow into six-steps and implements a three-layer architecture that combines orchestration, step skills, and atomic circuit skills. This design enables reusable, configurable, and extensible integration of capabilities from heterogeneous EDA agent projects. Empirical results on 41 hard VerilogEval v2 problems demonstrate both effectiveness and flexibility. In Experiment 1, single skill settings consistently improve over the baseline. Loop enabled settings raise Pass@1 from 0.000 to 0.805 and deliver up to 80.5% gain over gpt-5.2-codex under xhigh. In Experiment 2, cross-project skill compositions also reach 0.805 Pass@1. They provide up to 29.3% gain over the S1G4 baseline, outperform hierarchy-verilog by 14.6%, outperform VerilogCoder by 2.5%, and match MAGE. These findings show that modular skill construction is a practical direction for robust RTL automation. They also show that circuit skills built in LEGO can closely match and sometimes surpass the performance of original works through flexible composition.

VI. Future Work

As LLMs advance with stronger agentic capabilities, major platforms including Codex CLI [17], Claude Code [18], and OpenCode [19] have adopted skill-based architectures. Marketplaces like Skills.homes [31] now host thousands of daily updated skills. The skill paradigm offers natural advantages for EDA through standardization for cross-project reuse, composability for flexible workflows, and text-based interfaces that align with document heavy chip design tasks.

Acknowledgment

This project is supported in part by National Science and Technology Major Project (2021ZD0114702).

References

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

[2] N. Pinckney, C. Batten, M. Liu, H. Ren, and B. Khailany, "Revisiting verilogeval: Newer llms, in-context learning, and specification-to-rtl tasks," 2024. [Online]. Available: https://arxiv.org/abs/2408.11053

[3] 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)*, 2024, pp. 722–727.

[4] Z. Fang, R. Chen, Y. Guo, H. Dai, and L. Wang, "RTLBench: A multi-dimensional benchmark suite for evaluating llm-generated rtl code," in *2025 IEEE 43rd International Conference on Computer Design (ICCD)*, 2025, pp. 566–573.

[5] S. Liu, Y. Lu, W. Fang, M. Li, and Z. Xie, "OpenLLM-RTL: Open dataset and benchmark for llm-aided design rtl generation: Invited paper," in *2024 ACM/IEEE International Conference On Computer Aided Design (ICCAD)*, 2024, pp. 1–9.

[6] M. Liu, T.-D. Ene, R. Kirby *et al.*, "ChipNeMo: Domain-adapted llms for chip design," 2024. [Online]. Available: https://arxiv.org/abs/2311.00176

[7] B. Nadimi, G. O. Boutaib, and H. Zheng, "VeriMind: Agentic llm for automated verilog generation with a novel evaluation metric," 2025. [Online]. Available: https://arxiv.org/abs/2503.16514

[8] 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," *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 39, no. 1, pp. 300–307, Apr. 2025. [Online]. Available: https://ojs.aaai.org/index.php/AAAI/article/view/32007

[9] Z. Yu, M. Liu, M. Zimmer, Y. Celine, Y. Liu, and H. Ren, "Spec2RTL-Agent: Automated hardware code generation from complex specifications using llm agent systems," in *2025 IEEE International Conference on LLM-Aided Design (ICLAD)*, 2025, pp. 37–43.

[10] Y. Zhao, H. Zhang, H. Huang, Z. Yu, and J. Zhao, "MAGE: A multi-agent engine for automated rtl code generation," in *2025 62nd ACM/IEEE Design Automation Conference (DAC)*, 2025, pp. 1–7.

[11] Y. Zhao, D. Huang, C. Li, P. Jin, M. Song, Y. Xu, Z. Nan, M. Gao, T. Ma, L. Qi, Y. Pan, Z. Zhang, R. Zhang, X. Zhang, Z. Du, Q. Guo, and X. Hu, "CodeV: Empowering llms with hdl generation through multi-level summarization," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, pp. 1–1, 2025.

[12] 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*, vol. 44, no. 4, pp. 1448–1461, 2025.

[13] B. Kumar, S. Nanda, G. Parthasarathy, P. Patil, A. Tsai, and P. Choudhary, "HDL-GPT: High-quality hdl is all you need," 2024. [Online]. Available: https://arxiv.org/abs/2407.18423

[14] 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," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 29, no. 3, Apr. 2024. [Online]. Available: https://doi.org/10.1145/3643681

[15] F. Cui, C. Yin, K. Zhou, Y. Xiao, G. Sun, Q. Xu, Q. Guo, D. Song, D. Lin, X. Zhang, and Y. E. Liang, "OriGen: Enhancing rtl code generation with code-to-code augmentation and self-reflection," in *2024 ACM/IEEE International Conference On Computer Aided Design (ICCAD)*, 2024, pp. 1–9.

[16] 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," in *2024 IEEE 42nd International Conference on Computer Design (ICCD)*, 2024, pp. 162–169.

[17] OpenAI, "Codex CLI: Command-line interface for code agents," https://developers.openai.com/codex/cli/, 2024, version 0.98.0, Accessed: 2026-02-08.

[18] Anthropic, "Claude Code: Ai-powered coding assistant," https://www.anthropic.com/claude-code, 2024, accessed: 2026-02-08.

[19] OpenCode Contributors, "OpenCode: Open-source code generation platform," https://opencode.org, 2024, accessed: 2026-02-08.

[20] Icarus Verilog Project, "Icarus Verilog," https://steveicarus.github.io/iverilog/, 2026, accessed: 2026-02-07.

[21] Verilator Project, "Verilator," https://www.veripool.org/verilator/, 2026, accessed: 2026-02-07.

[22] W. Li, Y. Zou, C. Ellis, R. Purdy, S. Blanton, and J. M. F. Moura, "BRIDGES: Bridging graph modality and large language models within eda tasks," in *2025 IEEE International Conference on LLM-Aided Design (ICLAD)*, 2025, pp. 77–84.

[23] R. Qiu, G. L. Zhang, R. Drechsler, U. Schlichtmann, and B. Li, "AutoBench: Automatic testbench generation and evaluation using llms for hdl design," in *Proceedings of the 2024 ACM/IEEE International Symposium on Machine Learning for CAD*, ser. MLCAD '24. New York, NY, USA: Association for Computing Machinery, 2024. [Online]. Available: https://doi.org/10.1145/3670474.3685956

[24] J. Tang, J. Qin, K. Thorat, C. Zhu-Tian, Y. Cao, Y. K. Zhao, and C. Ding, "HiVeGen – hierarchical llm-based verilog generation for scalable chip design," in *2025 IEEE International Conference on LLM-Aided Design (ICLAD)*, 2025, pp. 30–36.

[25] D. Garcia-Gasulla, G. Kestor, E. Parisi, M. Albertí-Binimelis, C. Gutierrez, R. M. Ghorab, O. Montenegro, B. Homs, and M. Moreto, "TuRTLe: A unified evaluation of llms for rtl generation," in *2025 ACM/IEEE 7th Symposium on Machine Learning for CAD (MLCAD)*, 2025, pp. 1–12.

[26] S. Thakur, J. Blocklove, H. Pearce, B. Tan, S. Garg, and R. Karri, "AutoChip: Automating hdl generation using llm feedback," 2024. [Online]. Available: https://arxiv.org/abs/2311.04887

[27] H. Huang, Z. Lin, Z. Wang, X. Chen, K. Ding, and J. Zhao, "Towards llm-powered verilog rtl assistant: Self-verification and self-correction," 2024. [Online]. Available: https://arxiv.org/abs/2406.00115

[28] Y. Tsai, M. Liu, and H. Ren, "RTLFixer: Automatically fixing rtl syntax errors with large language models," in *2024 61st ACM/IEEE Design Automation Conference (DAC)*, 2024, pp. 1–6.

[29] J. Zhang, C. Liu, L. Cheng, X. Li, and H. Li, "Understanding and mitigating errors of llm-generated rtl code," 2026. [Online]. Available: https://arxiv.org/abs/2508.05266

[30] zjz1222, "Verilogassistant: Open-source reproduction repository," https://github.com/zjz1222/VerilogAssistant, 2026, gitHub repository, Accessed: 2026-02-08.

[31] Skills.homes, "Skills.homes: Agent skill marketplace," https://skills.homes/zh-CN, 2025, accessed: 2026-02-08.