REASONINGV: EFFICIENT VERILOG CODE GENERATION WITH ADAPTIVE HYBRID REASONING

Anonymous authors
Paper under double-blind review

ABSTRACT

Large Language Models (LLMs) have advanced Verilog code generation but still suffer from data quality, limited reasoning, and inefficiency. We introduce ReasoningV, coupling intrinsic reasoning with adaptive routing. Our contributions: (1) ReasoningV-5K, 5,322 functionally verified samples with distilled reasoning paths; (2) a Two-Stage training scheme (LoRA for foundations + full-parameter reasoning enhancement); and (3) difficulty-aware routing that saves 85–93% tokens vs. a strong commercial model and 32–75% vs. fixed-depth variants. On VerilogEval-human, RV-14B attains 73.9% pass@1; RV-7B reaches 57.8% with superior efficiency. Models, data, and code: https://github.com/BUAA-CLab/ReasoningV.

1 INTRODUCTION

Hardware Description Languages (HDLs) like Verilog are vital for digital design, yet translating natural-language specs into RTL remains challenging. Three hurdles dominate: **data quality** (unverified samples), **limited reasoning** for complex hardware tasks, and **inefficiency** from fixed-depth reasoning.

We present *ReasoningV*, which combines intrinsic reasoning with adaptive routing. It comprises: (1) **ReasoningV-5K**—5,322 functionally verified samples with distilled reasoning paths (CR=4.56); (2) **Two-Stage Training** that first builds foundations with LoRA then performs full-parameter reasoning enhancement; and (3) **Adaptive Reasoning** via a Judge Adapter that allocates tokens by difficulty, saving 85–93% vs commercial models and 32–75% vs fixed-depth variants.

On standard benchmarks, RV-14B reaches 73.9% pass@1 on VerilogEval-human and RV-7B 57.8%, demonstrating both effectiveness and efficiency.

2 RELATED WORK

2.1 VERILOG DATA AND MODEL CHALLENGES

LLM-based code generation struggles on HDLs due to parallel semantics, timing, and synchronization requirements Chowdhury et al. (2023); Pinckney et al. (2025). Existing datasets emphasize syntax or ranking, but rarely provide functional verification or reasoning traces, limiting structured reasoning learning. Table 1 summarizes typical resources; ReasoningV-5K contributes 5,322 simulation-verified triplets with distilled reasoning paths to fill these gaps.

Gap. Existing datasets often lack rigorous functional verification and do not provide explicit reasoning traces, which prevents models from learning structured problem decomposition for HDL generation. ReasoningV-5K fills both gaps by pairing simulation-verified solutions with distilled, code-free reasoning paths, enabling reliable supervision for hardware-aware reasoning.

2.2 VERILOG GENERATION MODELS

Data-driven Verilog generation aims to overcome traditional EDA limitations Compiler (2016). While recent works explore *training-free* prompting/agents (e.g., HDLCoRe Ping et al. (2025), VeriMind Nadimi et al. (2025)) and *training-based* finetuning or RL (e.g., CodeV Zhao et al. (2024),

Table 1: Verilog datasets for LLMs. ABV = Assertion-Based Verification; FPV = Formal Property Verification.

DATASET	SOURCE	SIZE	FEATURES
PyraNetNadimi et al. (2024)	GitHub+LLM	690K+	Syntax-checked, quality-ranked
MG-VerilogZhang et al. (2024)	Open-Source	11K+	Multi-grain descriptions
RTLCoderLiu et al. (2024)	LLM-gen	80K+	ABV/FPV subset available
VerilogEvalLiu et al. (2023)	HDLBits	330	Testbench evaluation
ReasoningV	PyraNet	5K	**Problem-solution-test triples**

VeriSeek Wang et al. (2025)), many still struggle with **limited reasoning capabilities** on complex hardware tasks, impacting functional correctness Liu et al. (2023); Lu et al. (2024). ReasoningV combines **intrinsic reasoning** (trained on the **ReasoningV-5K** dataset with a **Two-Stage** method) with **adaptive routing** at inference. It aims for greater robustness than prompt-elicited reasoning and more targeted enhancement than standard tuning or RL focused mainly on code structure or basic correctness.

Gap. Prior work either relies on training-free prompting (unstable across tasks) or task-agnostic finetuning (limited hardware reasoning depth). Few methods jointly strengthen intrinsic reasoning and enforce hardware-aware supervision. Our approach closes this gap by combining reasoning-enhanced data with staged training to build stable, verifiable Verilog reasoning capability.

2.3 Adaptive Reasoning

Enhancing LLM reasoning often incurs significant **computational inefficiency** if applied uniformly Wu et al. (2024), especially via methods like Chain-of-Thought (CoT) Ping et al. (2025). Efforts to improve efficiency include advanced prompting, architectural/training innovations, and adaptive inference techniques like dynamic routing or complexity-based resource allocation Wang et al. (2025); YangSui et al. (2025); Qu et al. (2025); Piskala et al. (2025). Building on these, ReasoningV introduces an **adaptive reasoning mechanism** (Sec 3.3) using a lightweight Judge Adapter to estimate problem complexity and dynamically allocate reasoning resources. This approach, related to complexity-based routing Piskala et al. (2025) but integrated with our trained reasoning capabilities, optimizes resource use across different problem difficulties in Verilog generation, achieving 85–93% token savings versus commercial reasoning models and 32–75% savings versus fixed-depth variants while maintaining performance.

Gap. Many efficiency-oriented techniques apply a fixed reasoning depth, over-spending on easy cases and under-spending on hard ones. Complexity-aware allocation for HDL generation remains under-explored. Our Judge Adapter provides difficulty-conditioned budgets with minimal overhead, improving throughput without sacrificing accuracy.

Table 2: Filtering stages and criteria for ReasoningV-5K construction.

Stage	Input	Output	Ratio	Filters
Compilable (Tier2)	65,344	65,344	1.00	Syntax/compile check (Icarus/Verilog parser)
Domain de-dup	65,344	21,000	0.32	Cosine $>$ 0.8, clique grouping, leave-one representative
Quality screening	21,000	7,000	0.33	Rule-guided checks (naming, FSM style, instantiation) via DeepSeek-R1 prompts
Functional verification	7,000	5,322	0.76	Simulation pass against paired testbench (Icarus)

3 Methodology

This section outlines the ReasoningV framework for Verilog code generation. As shown in Fig.1, our approach comprises three core components: constructing the ReasoningV-5K dataset, implementing a two-stage training method, and developing an adaptive reasoning mechanism. These elements systematically address poor data quality, limited reasoning capabilities, and computational inefficiency through data refinement, staged training, and dynamic reasoning optimization.

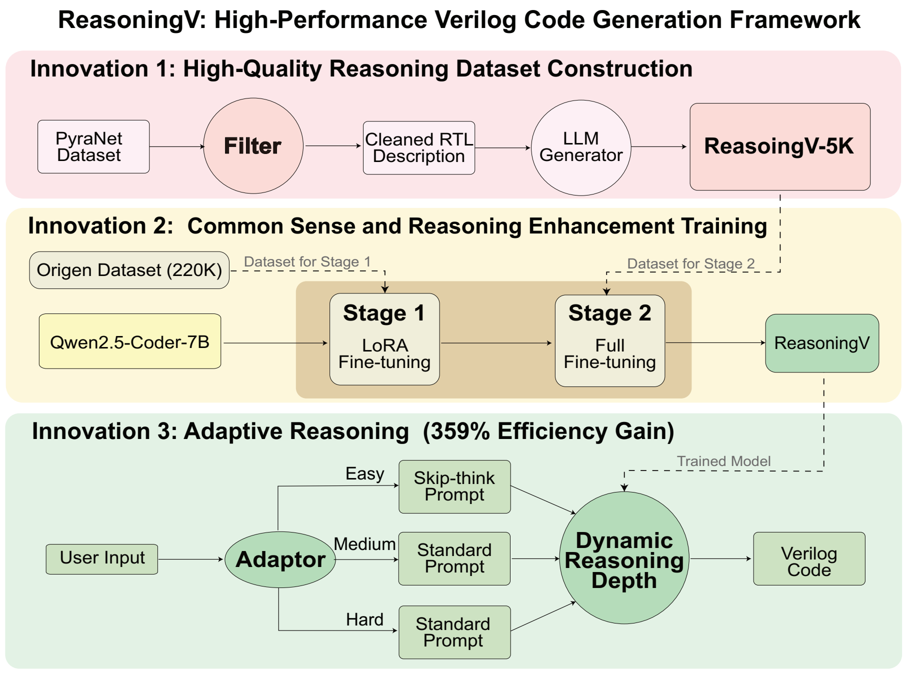


Figure 1: Overall architecture of the ReasoningV framework, highlighting intrinsic reasoning (trained via Two-Stage learning) and adaptive routing (Judge Adapter) for difficulty-aware generation.

3.1 High-quality reasoning dataset construction

High-quality training data is vital for effective HDL generation. We constructed ReasoningV-5K, a rigorously verified Verilog reasoning dataset containing 5,322 high-quality samples, through multi-stage filtering and verification processes designed to enhance data reliability and enforce design best practices. As illustrated in Fig. 2, our construction process consists of three main steps: (1) Multi-Dimensional Filtering of PyraNet samplesNadimi et al. (2024), (2) Reasoning Enhancement with DeepSeek-R1Guo et al. (2025), and (3) Final Dataset Assembly with functional testbench verification.

3.1.1 Multi-dimensional filtering process

PyraNet Dataset Selection and Initial Filtering. We started with the PyraNet dataset Nadimi et al. (2024) (approx. 690K samples), acknowledging its scale but also its limitations (redundancy, quality issues). Filtering was essential for quality and cost-effectiveness, aligning with findings that quality outweighs quantity in HDL training Muennighoff et al. (2025); Lu et al. (2024). We focused on the Tier2 subset, retaining only compilable samples (65,344 candidates).

Domain-Specific Redundancy Elimination. We classified candidates into 15 hardware domains (using Qwen2.5-32B with temperature=0 for deterministic decoding and a fixed label set) and applied a three-step redundancy elimination within each domain: (1) Cosine similarity calculation (threshold 0.8, empirically determined) for code and descriptions; (2) Maximal clique grouping using Bron-Kerbosch; (3) Selecting one representative per clique using a “leave-on” rule. Samples without a confident label were assigned to “Other.” This yielded approx. 21K unique samples.

Expert-Guided Automated Quality Assessment. We further refined the dataset to 7K samples using DeepSeek-R1, prompted with expert-defined rules covering naming conventions and RTL coding style best practices (e.g., module instantiation, state machine structure). Only samples conforming to these guidelines were retained, ensuring high code quality and readability.

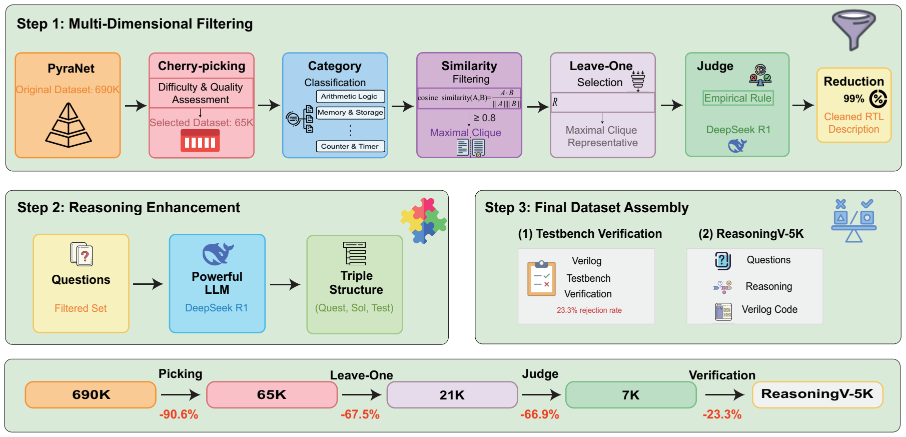
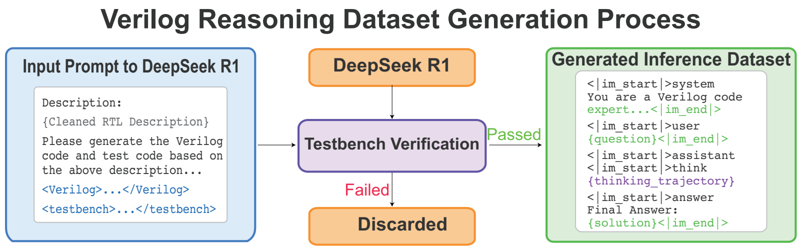


Figure 2: Multi-stage data filtering pipeline for ReasoningV-5K dataset construction.

3.1.2 Dataset verification and structure

We adopted a "problem-description-reasoning_path-solution-testbench" format. Each entry includes the problem description, a detailed reasoning path (distilled using DeepSeek-R1, see Fig 3), the Verilog solution, and a corresponding functional testbench (also generated with DeepSeek-R1 guidance).


Verilog Reasoning Dataset Generation Process
Input Prompt to DeepSeek R1
DeepSeek R1
Testbench Verification
Passed
Failed
Discarded
Generated Inference Dataset

```
Description:
{Cleaned RTL Description}
Please generate the Verilog
code and test code based on
the above description...
<Verilog>...</Verilog>
<testbench>...</testbench>
```

```
<|im_start|>system
You are a Verilog code
expert...<|im_end|>
<|im_start|>user
{question}<|im_end|>
<|im_start|>assistant
<|im_start|>think
{thinking_trajectory}
<|im_start|>answer
Final Answer:
{solution}<|im_end|>
```

Figure 3: Verilog Reasoning Path and Testbench Generation.

The reasoning paths capture the step-by-step thought process from problem understanding to solution design, enabling the model to learn structured hardware design reasoning. Figure 4 illustrates a concrete example of this reasoning process.

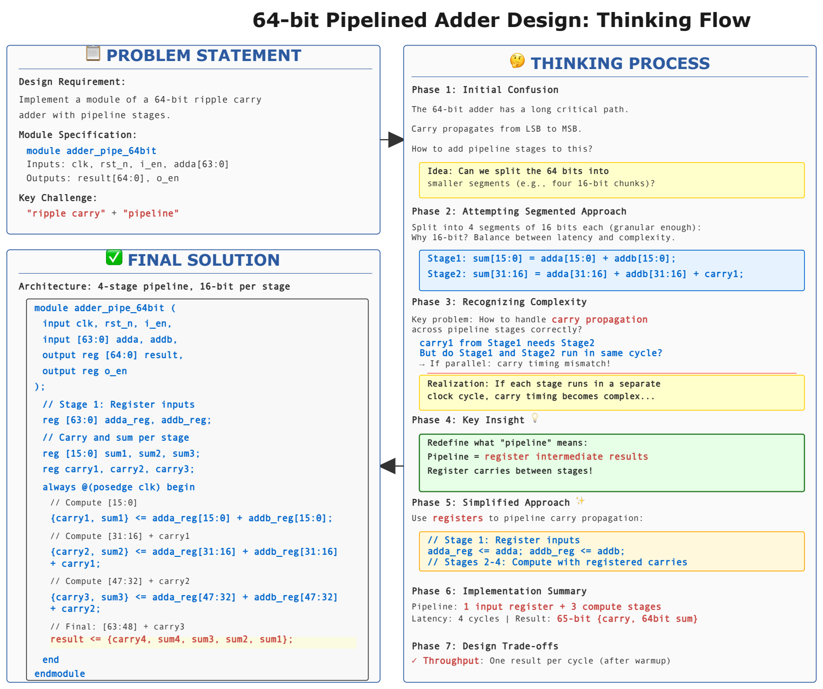
3.1.3 Reasoning path distillation details

Prompting. We elicit code-free reasoning with a fixed instruction (schema-only, no code). Full prompt and regex guards are provided in the appendix and repository.

Compression/distillation rules. We normalize the model's chain-of-thought into a concise, reproducible format:

- Strip all literal Verilog tokens, code blocks, and identifiers; retain only pseudocode-level structure.
- Enforce a stepwise schema (assumptions → I/O plan → states/datapath → corner cases).
- Cap length with a soft budget (max 512 tokens) and remove repetitive self-references.
- Keep only information derivable from the task text; drop any content that mirrors unit-test constants or golden outputs.

Leakage safeguards and reproducibility. To avoid training–test contamination and label leakage: (i) the reasoning path contains no final code or unit-test answers; (ii) regex-based guards reject outputs containing code-like tokens; (iii) evaluation benchmarks are excluded by exact-match filtering on prompts and metadata; and (iv) we release scripts to re-run distillation with the same prompt and budgets. Crucially, **functional verification** was performed by simulating each solution against its testbench using Icarus Verilog Williams & Baxter (2002). Samples failing simulation (approx. 23.3% of the 7K candidates) were discarded. This prioritization of functional accuracy resulted in the final ReasoningV-5K dataset: 5,322 high-quality, functionally verified instances with reasoning paths.


64-bit Pipelined Adder Design: Thinking Flow

PROBLEM STATEMENT

Design Requirement:

Implement a module of a 64-bit ripple carry adder with pipeline stages.

Module Specification:

```
module adder_pipe_64bit
Inputs: clk, rst_n, i_en, adda[63:0]
Outputs: result[64:0], o_en
```

Key Challenge:

"ripple carry" + "pipeline"

FINAL SOLUTION

Architecture: 4-stage pipeline, 16-bit per stage

```
module adder_pipe_64bit (
  input clk, rst_n, i_en,
  input [63:0] adda, addb,
  output reg [64:0] result,
  output reg o_en
);
  // Stage 1: Register inputs
  reg [63:0] adda_reg, addb_reg;
  // Carry and sum per stage
  reg [15:0] sum1, sum2, sum3;
  reg carry1, carry2, carry3;
  always @(posedge clk) begin
    // Compute [15:0]
    {carry1, sum1} <= adda_reg[15:0] + addb_reg[15:0];
    // Compute [31:16] + carry1
    {carry2, sum2} <= adda_reg[31:16] + addb_reg[31:16]
    + carry1;
    // Compute [47:32] + carry2
    {carry3, sum3} <= adda_reg[47:32] + addb_reg[47:32]
    + carry2;
    // Final: [63:48] + carry3
    result <= {carry4, sum4, sum3, sum2, sum1};
  end
endmodule
```

THINKING PROCESS

Phase 1: Initial Confusion

The 64-bit adder has a long critical path.

Carry propagates from LSB to MSB.

How to add pipeline stages to this?

Idea: Can we split the 64 bits into smaller segments (e.g., four 16-bit chunks)?

Phase 2: Attempting Segmented Approach

Split into 4 segments of 16 bits each (granular enough):
Why 16-bit? Balance between latency and complexity.

```
Stage1: sum[15:0] = adda[15:0] + addb[15:0];
Stage2: sum[31:16] = adda[31:16] + addb[31:16] + carry1;
```

Phase 3: Recognizing Complexity

Key problem: How to handle carry propagation across pipeline stages correctly?

carry1 from Stage1 needs Stage2
But do Stage1 and Stage2 run in same cycle?
→ If parallel: carry timing mismatch!

Realization: If each stage runs in a separate clock cycle, carry timing becomes complex...

Phase 4: Key Insight

Redefine what "pipeline" means:
Pipeline = register intermediate results
Register carries between stages!

Phase 5: Simplified Approach

Use registers to pipeline carry propagation:

```
// Stage 1: Register inputs
adda_reg <= adda; addb_reg <= addb;
// Stages 2-4: Compute with registered carries
```

Phase 6: Implementation Summary

Pipeline: 1 input register + 3 compute stages
Latency: 4 cycles | Result: 65-bit {carry, 64bit sum}

Phase 7: Design Trade-offs

✓ Throughput: One result per cycle (after warmup)

Figure 4: Reasoning flow example (problem → thinking → solution) for a 64-bit pipelined ripple-carry adder. The thinking process is a single module whose top aligns with the problem and bottom aligns with the solution, following the visual template used in our ReasoningV-5K dataset. This example demonstrates how complex hardware design challenges are systematically decomposed through structured reasoning phases before generating the final implementation.

Our filtering reduced the initial 690K samples by approx. 99.3%, significantly improving quality and ensuring functional correctness per sample.

3.2 Two-Stage Training

We adopt a concise two-stage scheme: (1) **Foundations**—train LoRA adapters (1.05% params) to acquire Verilog knowledge; (2) **Reasoning**—merge adapters and fine-tune all parameters on ReasoningV-5K to learn hardware reasoning patterns. This design yields stronger intrinsic reasoning than training-free prompting while preserving efficiency. Full hyperparameters and optimizer settings are provided in the appendix.

3.3 Adaptive Reasoning

We attach a lightweight Judge adapter (LoRA) to classify each task as Easy/Medium/Hard, then route decoding with matched prompts and token budgets: Easy (direct, 512), Medium (standard, 1280), Hard (extended, 4096). The Judge runs once per task; code is generated by the full model.

This conditional routing reduces tokens while preserving accuracy. Training details and data construction for the Judge are provided in the appendix.

4 REASONING EXAMPLE AND DATASET ANALYSIS

To address reviewer concerns on the clarity of our reasoning pipeline and the scientific basis of our dataset analysis, we provide concrete examples and transparent methodology.

4.1 A CONCRETE REASONING EXAMPLE

To illustrate how our reasoning-enhanced training translates into structured problem-solving, we include a compact, end-to-end example that demonstrates the transformation from problem statement through thinking process to final solution. Figure 4 shows the problem definition and final solution in one column, the complete thinking process in a separate column, with parallel arrows connecting them. This visualization follows the same template structure used in our ReasoningV-5K dataset, demonstrating how complex hardware design problems are decomposed into structured reasoning phases before generating the final Verilog code.

4.2 REASONINGV-5K DATASET DISTRIBUTION ANALYSIS

To provide transparency on the structure and coverage of ReasoningV-5K, we analyze the dataset distribution in the embedding space. Figure 5 projects item-level embeddings into 2D using Principal Component Analysis (PCA) and overlays density and trajectory cues for different categories and difficulty levels. This visualization reveals the dataset's internal structure, showing how different problem types cluster in the embedding space and demonstrating the diversity of our curated dataset.

The ReasoningV-5K dataset contains 5,322 functionally verified samples across 18 hardware design categories and three difficulty levels. The largest categories are "Basic Logic Units and Standard Cell Library" (2,104 samples, 39.5%), "Interface and Communication Protocols" (728 samples, 13.7%), "Arithmetic Units" (585 samples, 11.0%), and "Storage Elements and Memory" (499 samples, 9.4%). The difficulty distribution shows a bimodal mix with Easy (2,011 samples, 37.8%), Hard (2,154 samples, 40.5%), and Medium (1,157 samples, 21.7%), ensuring comprehensive coverage of both fundamental hardware components and complex design challenges.

4.3 METHODOLOGY AND REPRODUCIBILITY

Our dataset analysis follows a transparent and reproducible pipeline grounded in standard statistical visualization practices. The analysis pipeline is implemented in `code/plot_from_data.py` and consists of the following steps:

- **Data Loading:** We use utility functions `read_csv_column` and `load_plot_data` to systematically load and preprocess the ReasoningV-5K dataset, ensuring consistent data handling across all analyses.
- **Embedding Projection:** Text and image embeddings are projected to 2D using Principal Component Analysis (PCA), preserving maximum variance directions while enabling clear visualization. This dimensionality reduction maintains the essential geometric structure of the high-dimensional embedding space.
- **Visualization Generation:** The `plot_all` routine renders the full scatter plot and overlays per-category trends using smooth polylines. Points are grouped by category labels, and trajectory lines are computed to summarize structural patterns in the embedding space.
- **Reproducibility:** All hyperparameters (e.g., PCA components, smoothing window size, sample size) are fixed across runs. Figures are generated programmatically to ensure reproducibility and eliminate manual selection bias.

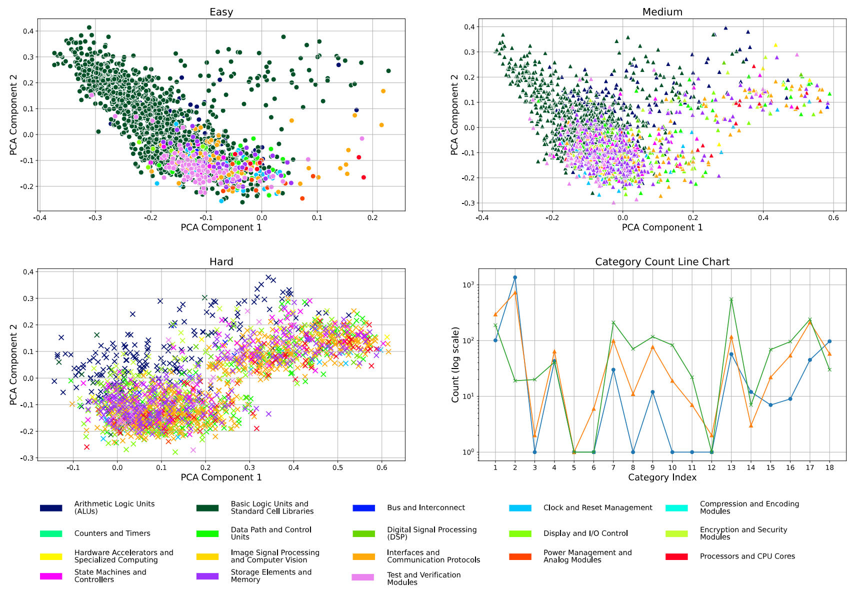


Figure 5: ReasoningV-5K distribution in a 2D PCA projection of text/image embeddings, highlighting category structure and trends. The visualization demonstrates the dataset's diversity and coverage across different problem types, with distinct clusters and trajectories for various hardware design categories.

This methodology ensures that the visual patterns observed in Figure 5—including cluster separation, category trajectories, and density distributions—arise from the underlying embedding geometry rather than manual selection or subjective placement. The systematic approach validates that ReasoningV-5K exhibits meaningful structure and diversity, supporting its effectiveness as a training dataset for reasoning-enhanced Verilog generation.

5 EXPERIMENTS

In this section, we evaluate ReasoningV's performance against baseline models and conduct detailed ablation studies to analyze the impact of different components in our framework. We first describe our experimental setup, including benchmarks and implementation details. Then, we present comparative results against some LLMs and models, and analyze ReasoningV's components through systematic ablation studies.

5.1 EXPERIMENTAL SETUP

Benchmarks: VerilogEval-human (156), VerilogEval-machine (143), RTLLM (30). Models: RV-7B (Qwen2.5-Coder-7B) and RV-14B (Qwen2.5-Coder-14B). Inference uses temperature 0.2, top_p 0.95; correctness is verified by Icarus Verilog Williams & Baxter (2002). We report pass@k with the standard unbiased estimator Kumar & Sharma (2023), drawing n=10 samples per problem and averaging across problems.

5.2 MAIN RESULTS

Table 3 presents a comprehensive comparison of ReasoningV against existing commercial LLMs, open-source models, and Verilog-specific models across all three benchmarks.

While large commercial models, particularly the reasoning-focused DeepSeek-R1 Guo et al. (2025), exhibit superior performance, ReasoningV significantly narrows the gap compared to other models of similar scale (7B, 14B parameters). RV-7B outperforms its base Qwen2.5-Coder-7B model by substantial margins: +23.5% on VerilogEval-human, +12.5% on VerilogEval-machine, and +13.3% on RTLLM for pass@1 accuracy, RV-14B outperforms its base Qwen2.5-Coder-14B model by sub-

Table 3: Performance comparison across Verilog code generation benchmarks. **VEval-H** = VerilogEval-human, **VEval-M** = VerilogEval-machine, **P@k** = pass@k, **DS** = DeepSeek, **CQ** = CodeQwen, – = not reported. Models marked with **bold** indicate best performance within their category.

Category	**Model**	**VEval-H**			**VEval-M**			**RTLLM**	
		P@1	P@5	P@10	P@1	P@5	P@10	P@1	P@5
Commercial	GPT-4o-mini	44.2	49.4	56.4	60.1	62.2	65.0	49.2	61.5
	DeepSeek-R1	81.7	88.1	89.7	81.1	85.8	87.4	58.5	66.8
	DeepSeek-V3	70.7	77.4	78.8	77.6	86.2	87.4	54.9	63.7
	Gemini-2.0	59.5	67.8	70.9	74.9	79.5	81.5	46.5	56.4
Open Source	Qwen-7B	34.3	46.3	50.0	61.1	75.7	78.6	31.3	45.2
	Qwen-14B	48.2	59.8	63.5	64.9	77.1	79.7	41.6	54.1
	Qwen-32B	47.6	59.0	61.5	61.5	64.1	75.5	40.2	55.7
Verilog	RTLCoder-DS	41.6	50.1	53.4	61.2	76.5	81.8	33.5	43.9
	BetterV-CQ	46.1	53.7	58.2	68.1	79.4	84.5		
	CraftRTL-15B	68.0	72.4		81.9	86.9		49.0	65.8
	OriGen-DS	54.4	60.1	64.2	74.1	82.4	85.7		65.5
	OriGen-LoRA	47.4	59.9	63.4	70.2	80.2	82.7	44.1	54.0
RV (Ours)	RV-7B	57.8	69.3	72.4	73.6	83.4	85.3	44.6	62.2
	RV-14B	73.9	83.7	85.9	75.6	85.2	87.4	50.6	64.7

stantial margins: +25.7% on VerilogEval-human, +10.7% on VerilogEval-machine, and +9% on RTLLM for pass@1 accuracy. This clearly demonstrates the effectiveness of our dataset curation and training methodology.

Key takeaways: (i) RV-14B achieves state-of-the-art performance among publicly available open-source models on VEval-H; (ii) RV-7B remains competitive with markedly lower token usage; (iii) gains are consistent across VEval-M and RTLLM.

The OriGen model Cui et al. (2024) represents another recent approach to improving Verilog generation through a two-LoRA architecture: Gen LoRA for initial code generation and Fix LoRA for error correction. For fair comparison of pure code generation capabilities (without self-reflection), we evaluated against OriGen using only its Gen LoRA component, which already achieves strong performance with a 47.4% pass@1 on VerilogEval-human through its code-to-code augmentation methodology. Despite OriGen's impressive results, RV-7B still outperforms it by 10.4 percentage points (57.8% vs. 47.4%), demonstrating the effectiveness of our approach that combines high-quality verified data (ReasoningV-5K) with full-parameter reasoning enhancement. This performance gap is particularly noteworthy as both models begin with 7B parameter base models but take different approaches to enhancement—ReasoningV's two-stage training with full-parameter optimization versus OriGen's parameter-efficient LoRA adaptation with data augmentation.

While concurrent works without public code release achieve higher performance (ScaleRTL-32B: 76.3-80.4%, CraftRTL-15B: 68.0%), RV-14B demonstrates the strongest results among fully accessible models with complete training methodology. Additionally, RV-14B achieves 75.6% pass@1 on VerilogEval-machine and 50.6% on RTLLM, demonstrating consistent strong performance across benchmarks. RV-7B achieves 57.8% pass@1 on VerilogEval-human, demonstrating strong performance and scalability.

5.3 Ablation studies

We conducted comprehensive ablation studies to analyze the impact of different components in the ReasoningV framework, focusing on the effectiveness of the two-stage training methodology and the adaptive reasoning mechanism.

5.3.1 Effectiveness of Two-Stage training

Table 4 presents the results of our ablation study on the training methodology, comparing models trained with different stages of our proposed approach against the base model.

Table 4: Ablation study on the effectiveness of two-stage training methodology. **CT** = Commonsense Training, **RET** = Reasoning Enhancement Training.

Model	VerilogEval-human	VerilogEval-machine	RTLLM
Base	34.3	61.1	31.3
RV-CT	51.3	64.5	42.5
RV-RET	45.5	65.7	33.3
RV-7B	**57.8**	**73.6**	**44.6**

RV-CT (Stage 1 Only) with LoRA on OriGen significantly improves performance across all benchmarks compared to the base model, confirming effective foundational knowledge establishment.
RV-RET (Stage 2 Only) with full-parameter training on ReasoningV-5K shows improvement but underperforms RV-CT on VerilogEval-human and RTLLM pass@1, indicating reasoning alone cannot compensate for insufficient domain knowledge.
RV-7B(Stage 1 + Stage 2) achieves superior performance across all benchmarks with substantial gains over single-stage models, validating that the sequential approach produces optimal results. In particular, the complementary roles of commonsense grounding and reasoning enhancement are synergistic rather than substitutive, and the improvements generalize across datasets.

5.3.2 Analysis of adaptive reasoning mechanism

Table 5 presents a comparison of different reasoning approaches applied to our RV-7B model. We include fixed difficulty settings (forcing Easy: direct generation, Medium: standard reasoning, or Hard: extended reasoning mode) and our proposed adaptive reasoning mechanism. The table shows the average number of generated tokens per problem for each approach across the benchmarks, with percentage changes relative to the adaptive mode. Additionally, results for DeepSeek-R1 are provided for reference.

Table 5: Performance and efficiency of reasoning modes on Verilog benchmarks.

MODE	VerilogEval-H		VerilogEval-M		RTLLM	
	P@1	Tokens	P@1	Tokens	P@1	Tokens
DS-R1	81.7	6070 (+584%)	81.1	3311 (+680%)	58.5	12686 (+1377%)
Easy	32.1	**235** (-74%)	51.3	**164** (-61%)	29.7	**356** (-59%)
Medium	54.3	1302 (+47%)	68.4	1156 (+173%)	42.8	1650 (+92%)
Hard	**57.8**	2473 (+178%)	**73.6**	1725 (+307%)	44.6	3940 (+359%)
Adaptive(7B)	53.0	888	**73.6**	424	**45.4**	859

Note: **H** = human, **M** = machine, P@1 denotes pass@1 metric, **DS** = DeepSeek. Token percentages are relative to Adaptive (baseline).

The results highlight the significant efficiency benefits of our adaptive reasoning mechanism:
Token Efficiency: RV-Adaptive achieves substantial reductions in average token consumption compared to both commercial models and our forced reasoning modes. Compared to DeepSeek-R1, our adaptive approach uses 85% fewer tokens on VerilogEval-human, 87% fewer on VerilogEval-machine, and 93% fewer on RTLLM, while still delivering competitive performance. Even within our own model variants, RV-Adaptive demonstrates remarkable efficiency, using 32% fewer tokens than Forced Medium and 64% fewer tokens than Forced Hard across all benchmarks. The savings are even more pronounced on VerilogEval-machine (63% vs. Forced Medium, 75% vs. Forced Hard) and RTLLM (48% vs. Forced Medium, 78% vs. Forced Hard).
Performance Trade-off: RV-Adaptive(7B) demonstrates an effective balance between performance and efficiency. On VerilogEval-human, there is a modest pass@1 decrease of 4.8% compared to always using the Hard mode (Forced Hard), but it still significantly outperforms the base model and RV-CT. Notably, on the more structured VerilogEval-machine benchmark, RV-Adaptive achieves the *same* pass@1 performance as Forced Hard while using 75% fewer tokens. On RTLLM, RV-Adaptive slightly outperforms Forced Hard in pass@1 (45.4% vs 44.6%), potentially due to more appropriate reasoning allocation.

6 Discussion and Conclusion

Our experimental results demonstrate substantial advances: RV-14B achieves 73.9% pass@1 on VerilogEval-human and RV-7B achieves 57.8%, while the adaptive approach attains 85–93% token savings versus commercial reasoning models and 32–75% versus fixed-depth variants with maintained accuracy. These results validate the ReasoningV framework: verified data, Two-Stage training, and adaptive routing collectively improve Verilog generation quality and efficiency. ReasoningV systematically addressed three critical challenges in hardware design automation: the deficiency of high-quality training data, limited reasoning capabilities for complex hardware design tasks, and computational inefficiency during inference. The large performance gain observed when moving from the base model to RV-14B (+39.6% on VerilogEval-human) highlights the critical impact of our approach. The ablation studies confirm that both training stages contribute substantially to the final performance, with the first stage establishing essential foundational knowledge and the second stage enhancing reasoning capabilities.

Our experimental evaluations demonstrated that ReasoningV substantially advances the SOTA for open-source Verilog generation models. RV-14B achieved 73.9% pass@1 on VerilogEval-human, 75.6% pass@1 on VerilogEval-machine, and 50.6% pass@1 on RTLLM, outperforming previous open-source models by significant margins. Notably, RV-14B surpassed not only its base model but also much larger models like Qwen2.5-Coder-32B, demonstrating that our targeted approach can be more effective than simply scaling model size. The adaptive reasoning mechanism offers a practical solution to the computational inefficiency problem, significantly reducing resource requirements (85–93% token savings versus commercial reasoning models, 32–75% versus fixed-depth variants) while maintaining strong performance. The benchmark-specific adaptive performance suggests that our Judge Adapter effectively identifies when extensive reasoning is necessary versus when a more direct approach suffices. Notably, for the more structured VerilogEval-machine benchmark, adaptive reasoning achieves identical performance to the full Hard mode while using only a quarter of the tokens.

Despite these advances, several limitations remain. The ReasoningV-5K dataset, while high-quality, is relatively small and may not cover the full diversity of hardware design challenges. The functional verification employed ensures behavioral correctness but does not guarantee synthesizability or timing compliance. Additionally, there is still a performance gap compared to top commercial models, suggesting room for further improvement.

Future work could explore several promising directions: (1) scaling the ReasoningV-5K dataset while maintaining its quality standards; (2) applying the ReasoningV methodology to larger base models, particularly the 14B parameter models that showed promising scaling characteristics; (3) developing more sophisticated adaptive reasoning mechanisms with finer-grained complexity assessment and budget allocation; (4) integrating feedback from downstream EDA tools into the training and inference process; and (5) extending the approach to support other HDLs and complex design tasks.

ReasoningV represents a significant step toward more reliable, capable, and efficient AI tools for hardware design automation. By addressing the foundational challenges of data quality, reasoning depth, and computational efficiency through targeted methodological innovations, our work provides a pathway for harnessing the potential of LLMs to assist in the increasingly complex task of hardware design. The ReasoningV model, dataset, and associated code are publicly available to foster further research and development in this rapidly evolving field.

References

Animesh B Chowdhury, Shailja Thakur, Hammond Pearce, Ramesh Karri, and Siddharth Garg. Towards the imagenets of ml4eda. In *2023 IEEE/ACM International Conference on Computer Aided Design (ICCAD)*, pp. 1–7. IEEE, 2023.

Synopsys Design Compiler. Synopsys design compiler. *Pages/default. aspx*, 2016.

Fan Cui, Chenyang Yin, Kexing Zhou, Youwei Xiao, Guangyu Sun, Qiang Xu, Qipeng Guo, Demin Song, Dahua Lin, Xingcheng Zhang, et al. Origen: Enhancing rtl code generation with code-to-code augmentation and self-reflection. *arXiv preprint arXiv:2407.16237*, 2024.

Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Ruoyu Zhang, Runxin Xu, Qihao Zhu, Shirong Ma, Peiyi Wang, Xiao Bi, et al. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. *arXiv preprint arXiv:2501.12948*, 2025.

Aman Kumar and Priyanka Sharma. Open ai codex: An inevitable future? *International Journal for Research in Applied Science and Engineering Technology*, 11:539–543, 2023.

Mingjie Liu, Nathaniel Pinckney, Brucek Khailany, and Haoxing Ren. Verilogeval: Evaluating large language models for verilog code generation. In *2023 IEEE/ACM International Conference on Computer Aided Design (ICCAD)*, pp. 1–8. IEEE, 2023.

Shang Liu, Wenji Fang, Yao Lu, Jing Wang, Qijun Zhang, Hongce Zhang, and Zhiyao Xie. Rtlcoder: Fully open-source and efficient llm-assisted rtl code generation technique. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2024.

Yao Lu, Shang Liu, Qijun Zhang, and Zhiyao 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)*, pp. 722–727. IEEE, 2024.

Niklas Muennighoff, Zitong Yang, Weijia Shi, Xiang Lisa Li, Li Fei-Fei, Hannaneh Hajishirzi, Luke Zettlemoyer, Percy Liang, Emmanuel Candès, and Tatsunori Hashimoto. s1: Simple test-time scaling. *arXiv preprint arXiv:2501.19393*, 2025.

Bardia Nadimi, Ghali Omar Boutaib, and Hao Zheng. Pyranet: A multi-layered hierarchical dataset for verilog. *arXiv preprint arXiv:2412.06947*, 2024.

Bardia Nadimi, Ghali Omar Boutaib, and Hao Zheng. Verimind: Agentic llm for automated verilog generation with a novel evaluation metric. *arXiv preprint arXiv:2503.16514*, 2025.

Nathaniel Pinckney, Christopher Batten, Mingjie Liu, Haoxing Ren, and Brucek Khailany. Revisiting verilogeval: A year of improvements in large-language models for hardware code generation. *ACM Transactions on Design Automation of Electronic Systems*, 2025.

Heng Ping, Shixuan Li, Peiyu Zhang, Anzhe Cheng, Shukai Duan, Nikos Kanakaris, Xiongye Xiao, Wei Yang, Shahin Nazarian, Andrei Irimia, and Paul Bogdan. Hdlcore: A training-free framework for mitigating hallucinations in llm-generated hdl, 2025. URL https://arxiv.org/abs/2503.16528.

Deepak Babu Piskala, Vijay Raajaa, Sachin Mishra, and Bruno Bozza. Dynamic llm routing and selection based on user preferences: Balancing performance, cost, and ethics. *arXiv preprint arXiv:2502.16696*, 2025.

Xiaoye Qu, Yafu Li, Zhaochen Su, Weigao Sun, Jianhao Yan, Dongrui Liu, Ganqu Cui, Daizong Liu, Shuxian Liang, Junxian He, et al. A survey of efficient reasoning for large reasoning models: Language, multimodality, and beyond. *arXiv preprint arXiv:2503.21614*, 2025.

Ning Wang, Bingkun Yao, Jie Zhou, Xi Wang, Zhe Jiang, and Nan Guan. Large language model for verilog generation with code-structure-guided reinforcement learning, 2025. URL https://arxiv.org/abs/2407.18271.

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

Siwei Wu, Zhongyuan Peng, Xinrun Du, Tuney Zheng, Minghao Liu, Jialong Wu, Jiachen Ma, Yizhi Li, Jian Yang, Wangchunshu Zhou, Qunshu Lin, Junbo Zhao, Zhaoxiang Zhang, Wenhao Huang, Ge Zhang, Chenghua Lin, and J. H. Liu. A comparative study on reasoning patterns of openai's o1 model, 2024. URL https://arxiv.org/abs/2410.13639.

Yu-NengChuang YangSui, JiamuZhang GuanchuWang, JiayiYuan TianyiZhang, AndrewWen HongyiLiu, Shaochen Henry Zhong, and XiaHu HanjieChen. Stop overthinking: A survey on efficient reasoning for large language models. *arXiv preprint arXiv:2503.16419*, 2025.

Yongan Zhang, Zhongzhi Yu, Yonggan Fu, Cheng Wan, and Yingyan Celine Lin. Mg-verilog: Multi-grained dataset towards enhanced llm-assisted verilog generation. In *2024 IEEE LLM Aided Design Workshop (LAD)*, pp. 1–5. IEEE, 2024.

Yang Zhao, Di Huang, Chongxiao Li, Pengwei Jin, Ziyuan Nan, Tianyun Ma, Lei Qi, Yansong Pan, Zhenxing Zhang, Rui Zhang, et al. Codev: Empowering llms for verilog generation through multi-level summarization. *arXiv preprint arXiv:2407.10424*, 2024.

Appendix: Reproducibility Details

A. Reasoning path prompt (code-free)

We elicit structured, code-free reasoning using the following template (schema only):

```
You are a hardware design expert. Given a Verilog design task:
- Decompose the problem (interfaces, states, datapath, timing).
- Provide ONLY the reasoning path in structured bullets.
- Do NOT output any Verilog code.
Problem:
<natural language spec here>
Output:
1) Assumptions/constraints
2) I/O and state plan (no code)
3) Module decomposition and signal plan (no code)
4) Corner cases and test intent (no code)
```

B. Regex guards (anti-leakage)

We reject generations containing code-like tokens via regex, e.g.,

- `(?i)\bmodule\b|\bendmodule\b|assign\b|always\b|posedge|negedge`
- Backticked code blocks or patterns resembling HDL declarations (e.g., `;\s*$`).

Benchmarks are excluded by exact-match filters on prompts/metadata.

C. Training hyperparameters

Two-Stage: Stage 1 (Foundations) trains LoRA adapters (rank=32, $\alpha = 32$; 1.05% params) on OriGen (220K, seq 2,048, 5 epochs). Stage 2 (Reasoning) merges adapters and fine-tunes all params on ReasoningV-5K (seq 8,192, 5 epochs) with AdamW, bf16, and FSDP. Learning-rate/optimizer details and layer targets are provided in the released scripts.

Judge adapter: 10,000-question difficulty dataset distilled from DeepSeek-V3; LoRA training with temperature 0 decoding and a fixed label set (Easy/Medium/Hard).

D. Inference and evaluation

Decoding uses temperature 0.2, top_p 0.95. We draw n=10 samples and compute pass@k via the unbiased estimator; correctness is verified using Icarus Verilog. Example commands and seeds are included in the repository (eval scripts and configs).