HAVEN: Hallucination-Mitigated LLM for Verilog Code Generation Aligned with HDL Engineers

Yiyao Yang*§, Fu Teng†§, Pengju Liu*§, Mengnan Qi*, Chenyang Lv*, Ji Li‡, Xuhong Zhang† Zhezhi He*
*School of Electronic Information and Electrical Engineering, Shanghai Jiao Tong University, Shanghai, China
†School of Software Technology, Zhejiang University, Ningbo, China; ‡Independent Researcher
§These authors contributed equally
{yangyiyao, zhezhi.he}@sjtu.edu.cn

***Abstract*—Recently, the use of large language models (LLMs) for Verilog code generation has attracted great research interest to enable hardware design automation. However, previous works have shown a gap between the ability of LLMs and the practical demands of hardware description language (HDL) engineering. This gap includes differences in how engineers phrase questions and hallucinations in the code generated. To address these challenges, we introduce HAVEN, a novel LLM framework designed to mitigate hallucinations and align Verilog code generation with the practices of HDL engineers. HAVEN tackles hallucination issues by proposing a comprehensive taxonomy and employing a chain-of-thought (CoT) mechanism to translate symbolic modalities (*e.g.* truth tables, state diagrams, *etc.*) into accurate natural language descriptions. Furthermore, HAVEN bridges this gap by using a data augmentation strategy. It synthesizes high-quality instruction-code pairs that match real HDL engineering practices. Our experiments demonstrate that HAVEN significantly improves the correctness of Verilog code generation, outperforming state-of-the-art LLM-based Verilog generation methods on VerilogEval and RTLLM benchmark. HAVEN is publicly available at https://github.com/Intelligent-Computing-Research-Group/HaVen.**

***Index Terms*—Verilog Code Generation, LLM**

TABLE I: **LLM-generated vs. HDL engineer-used formats**

Format generated by LLM	Format used by HDL engineers			
	Current State	**Input (x)**	**Next State**	**Output (out)**
"Implement the state machine with a combinational always block, which is used to determine the next state based on the current state and the values of the x port. If the current state is A and the x port is low, then..."	A	0	B	0
	A	1	A	0
	B	0	A	1
	B	1	B	1

I. INTRODUCTION

Large language models (LLMs) have achieved remarkable success in generating code for software programming languages like Python [1]–[6], benefiting from extensive documentation and rich training datasets [7]. However, fine-tuning LLMs for hardware description language (HDL) generation, particularly Verilog, presents distinct challenges that set it apart from software code generation [8]–[10]. These challenges arise from several key factors elaborated as follows.

Hallucination [11] in language models refers to the generation of incorrect or misleading information not grounded in reality. While prior researches, such as CodeHalu [12] and HalluCode [13], propose taxonomies for hallucination and analyze this phenomenon in the context of Python code generation. These studies mostly focus on Python-specific error analysis. In contrast, HDLs exhibit unique characteristics, *e.g.*, their hardware-centric nature, timing requirements, and concurrency models, making them fundamentally different from software languages like Python [14]. As a result, LLMs often lack the necessary domain-specific knowledge, resulting in frequent hallucinations, especially when LLMs attempt to generate Verilog code that aligns with the practices of HDL engineers[1]. They may produce syntax, functional or logical errors that do not match the expectation of HDL engineers. Previous works like RTLFixer [16] and OriGen [17] have focused on *enhancing LLMs by reducing syntax errors*. However, these methods fall short in providing the *diverse, domain-rich Verilog samples needed to capture HDL-specific nuances*, and thus do not effectively mitigate hallucinations.

The practices of HDL engineers refer to standardized formats for describing functionality and problems, along with digital design conventions crucial for well-structured code implementation [18], [19]. The scarcity of high-quality datasets aligned with these practices is the primary reason for hallucinations and the lack of domain-specific knowledge in LLMs. Fine-tuning LLMs for Verilog code generation requires datasets consisting of rich instruction-code pairs, which are currently scarce. Although open-source repositories [20] contain Verilog code snippets, these are often devoid of comprehensive natural language descriptions. Existing solutions, including RTLCoder [21], OriGen [17], and AutoVCoder [22], have attempted to generate data by using closed-source LLMs to add descriptions to open-source Verilog code samples. However, this approach has notable limitations:

- **Trivial and Misaligned Descriptions:** Closed-source LLMs are not specialized in HDL. The descriptions generated often fail to capture the complexities of real-world register-transfer level (RTL) design scenarios, as shown in Table I.
- **Hallucinations in Fine-Tuned Models:** Fine-tuning LLMs on such misaligned data can result in models that are prone to hallucination in practical RTL design scenarios. LLMs fail to realize the practical format as shown in Table I.

In this paper, we present HAVEN, *a novel LLM framework specifically designed to mitigate hallucinations and align Verilog code generation with the practices of HDL engineers*. The unique challenges in HDL generation, particularly for

This work is partially supported by National Key R&D Program of China (2022YFB4500200), National Natural Science Foundation of China (No.62102257). Corresponding author: Zhezhi He.

[1]Recently, the popular VerilogEval benchmark, recognizing the same issue, proposed a new version that focuses on human evaluation problems. [15]

TABLE II: **Taxonomy of Hallucination in LLM-based Verilog Code Generation**

Type	Sub-types	Example prompt causing hallucination	Incorrect code	Error analysis

Symbolic Hallucination	State Diagram Misinterpretation	Implement this FSM...
		A[out=0]–[in==0]–>B...
		A[out=0]–[in==1]–>A...
			`case(state)`
			`A: begin`
			`if(in) next_state = B;`
			`else next_state = A;...`
				"A" and "B" should be reversed.

	Waveform Chart Misinterpretation	Implement the waveforms below... a: 0 1 0 1 ... b: 0 0 1 1 ... out: 0 0 0 1 ...	... `assign out = a + b;` ...	"out" should be "a & b".
	Truth Table Misinterpretation	Implement the truth table below... a | b | out 1 | 1 | 1 ... 1 | 0 | 0 ...	... `assign out = a | b;` ...	"out" should be "a & b". (Due to limited space, truth table is partially omitted)

Knowledge Hallucination	Digital Design Convention Misapplication	Implement a digit detector, using conventional FSM ...	`case(state)`
			`A: state=a+b; ...`
				"state" should be "next_state". A conventional FSM should contain "state transition", "next_state logic", "output logic".

	Verilog Syntax Misapplication	Implement a 4-bit adder ...	`def adder_4bit()` ...	The definition of module is syntactically wrong. "def" should be "module"

	Misunderstanding Verilog-specific Attributes	Implement this module using asynchronous reset signal	`always@(posedge clk)`
			`if(!reset)...`
				The reset should be asynchronous.

Logical Hallucination	Incorrect Logical Expression	Create a module, the output signal equals a plus b, then or c.	`assign output = (a + c) & b;` ...	"output" signal should be "(a+b)|c".

	Incorrect Handling of Corner Cases	Implement logic of two inputs. Output equals 1 when a and b are both 1, otherwise 0.	`case({a, b})`
			`2'b11: out = 1;`
			`endcase`
				"default" case is ignored.

	Failure in Adhering to Instructional Logic	Implement the logic below:
		if a == 0 && b == 0; out = 0;
		elif a == 1 && b == 0; out = 0...
			`if(a==0 || b==0) out = 0;`
			`else if(a==1 && b==0) out = 0;`
			...
				The first "if" expression should be "a==0 && b==0".

†: `code` in red highlights the error raised during the code generation.

Verilog, stem from its symbolic nature, knowledge requirements, and the logical complexities inherent in practical design scenarios. HAVEN addresses these challenges through a three-stage methodology focused on reducing hallucinations in LLM-generated code, while enhancing adherence to the practices of HDL engineers. Our key contributions can be summarized as:

▷ **Mitigating Hallucinations Guided by a Comprehensive Taxonomy:** We introduce a detailed hallucination taxonomy tailored to Verilog code generation (Section II), encompassing symbolic, knowledge, and logical hallucinations. Using a chain-of-thought (CoT) mechanism (Section III-B), HAVEN systematically transforms symbolic modalities (*e.g.*, truth tables and state diagrams) into natural language descriptions, guiding the LLM to generate more accurate Verilog code and reducing symbolic hallucination. Besides, we synthesize data that inject domain-specific knowledge (Section III-C) and enhance logical reasoning ability (Section III-D), addressing knowledge and logical hallucinations.

▷ **Aligning Code Generation with the Practices of HDL Engineers:** To the best of our knowledge, HAVEN is the first endeavor to bridge the gap between closed-source LLM synthesized data and HDL engineers. We employ a data augmentation strategy that synthesizes high-quality instruction-code pairs by incorporating manual exemplars and synthesized data reflecting key aspects of Verilog design. This enriched dataset ensures that fine-tuned LLMs not only adhere to syntactical correctness, but also align with the intricate requirements of practical HDL engineering scenarios.

▷ **Boosting Verilog Code Generation Performance:** HAVEN significantly improves the performance of LLMs in Verilog code generation tasks, achieving up to a 6.7% increase in pass@1 and a 4.7% increase in pass@5 compared to competing models on VerilogEval(v1)-Human. By mitigating hallucinations and aligning code generation with the practices of HDL engineers, HAVEN enhances the reliability and accuracy of LLM-generated Verilog code, making it more suitable for the use cases of prompts generated by engineers.

II. Taxonomy of Hallucination

We identify and categorize the hallucinations in threefold, *i.e.*, *symbolic hallucination, knowledge hallucination, and logical hallucination*, as tabulated in Table II.

Symbolic Hallucination occurs when LLMs encounter symbols, diagrams, or formats that are not directly interpretable. LLMs often fail to extract information correctly from these symbolic modalities, causing incorrect or incomplete code generation. The symbolic hallucination includes the misinterpretation of state diagrams, waveform charts and truth tables.

Knowledge Hallucination arises when LLMs lack domain-specific knowledge, leading to irrelevant or incorrect content generation, which can be further specified as:

☐ Digital Design Convention Misapplication: LLMs are provided with clear module descriptions, but the generated code is poorly structured or incomplete, violating conventions of HDL engineers for commonly used Verilog modules.

☐ Verilog Syntax Misapplication: LLMs incorrectly use key language constructs, applying them in ways that do not match their intended functionality or meaning.

☐ Misunderstanding Verilog-specific Attributes: LLMs struggle to handle the unique characteristics and complex features of Verilog, *e.g.* different reset mechanisms or edge-triggering modes.

Logical hallucination occurs when LLMs lack logical reasoning ability, consequently the output does not follow with the given instructions, which can be further specified as:

☐ Incorrect Logical Expression: LLM attempts logical reasoning but produces expressions that do not accurately reflect the requirements of the task, often oversimplifying the logic or failing to satisfy the specific conditions.

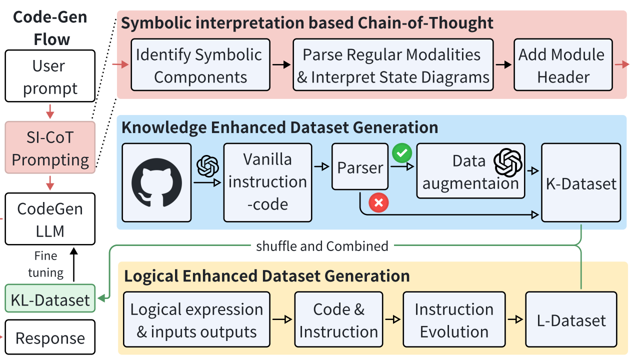

Code-Gen Flow
User prompt
SI-CoT Prompting
CodeGen LLM
Fine tuning
KL-Dataset
Response
Symbolic interpretation based Chain-of-Thought
Identify Symbolic Components
Parse Regular Modalities & Interpret State Diagrams
Add Module Header
Knowledge Enhanced Dataset Generation
Vanilla instruction -code
Parser
Data augmentaion
K-Dataset
shuffle and Combined
Logical Enhanced Dataset Generation
Logical expression & inputs outputs
Code & Instruction
Instruction Evolution
L-Dataset

Fig. 1: **Overview of HAVEN framework**.

TABLE III: **Examples of SI-CoT Interpretation**

Instructions before Interpretation		**Instructions after Interpretation**

A[out=0]–[x=0]->B
A[out=0]–[x=1]->A
B[out=1]–[x=0]->A
B[out=1]–[x=1]->B
	LLM
	⇒
		States&Outputs: 1.state A(out=0); 2. state B(out=1)
		State transition:
		1. From state A: If x = 0, then transit to state B;
		If x = 1, then transit to state A
		2. From state B: If x = 0; then transit to state A;
		If x = 1, then transit to state B

a | b | out 0 | 0 | 0 0 | 1 | 0 1 | 0 | 0 1 | 1 | 1	Parser ⇒	**Variables**: 1. a(input); 2. b(input); 3. out(output) **Rules**: 1. If a=0, b=0, then out=0; 2. If a=0, b=1, then out=0; 3. If a=1, b=0, then out=0; 4. If a=1, b=1, then out =1;
a: 0 1 1 0... b: 1 0 1 0... out: 1 0 0 1... time(ns): 0 10 20 30 ...	Parser ⇒	**Variables**: 1. a(input); 2. b(input); 3. out(output) **Rules**: When time is 0ns, a=0, b=1, out=1; When time is 10ns, a=1, b=0. out=0...

- ☐ Incorrect handling of corner cases: LLM generates code that fails to handle corner cases or undefined conditions.
- ☐ Failure in adhering to instructional logic: LLM is expected to faithfully implement the logic described in the instructions but fails, resulting in hallucinations.

III. Methodology

A. *Overview of* HAVEN

The framework overview of HAVEN is illustrated in Fig. 1. Given a user prompt, *i.e.*, instructions for Verilog code generation (CodeGen), it will be fed into a *CoT prompting model* to translate the vanilla user prompt into a refined CoT prompt, which can mitigate symbolic hallucinations. Then, the CoT prompt is sent to the *CodeGen-LLM* for an end-to-end inference. In our experiments, we use the same pre-trained models for both *CoT prompting model* and *CodeGen-LLM*.

As countermeasures against the hallucination taxonomy, HAVEN integrates three techniques, *i.e.*, *symbolic interpretation based CoT* (SI-CoT) to mitigate symbolic hallucination, while fine-tuning the *CodeGen-LLM* with *knowledge enhanced dataset* (K-dataset) and *logical enhanced dataset* (L-dataset) against knowledge- and logical-hallucinations respectively.

B. *Symbolic Interpretation based Chain-of-Thought*

SI-CoT leverages structured reasoning to convert symbolic inputs (*i.e.*, state diagrams, waveform charts and truth tables), into an legible form for *CodeGen-LLM*s. This approach addresses the challenge of symbolic hallucination, ensuring that the interpretations of input instructions remain aligned with their intended functional descriptions. SI-CoT applies incremental reasoning steps, allowing LLMs to process each input modality systematically, which in Fig. 1 can be further specified as:

❶ **Identify Symbolic Components.** The first step involves *CoT prompting model* recognizing whether the input contains symbolic representations. The process proceeds with interpretation, unless the input consists solely of natural language.

❷ **Parse Regular Modalities and Interpret State Diagrams.** Waveform charts and truth tables are regular modalities that can be directly processed by a parser. Once they are identified, an external parser is used to extract relevant data from the representation, then structure them into a uniform instruction format that can be interpreted by the *CodeGen-LLM*. If the identified symbolic component is state diagram, we adopt *CoT prompting model* interprets the diagram by translating its elements into a concise natural language description. This description includes the detailed relationships between states, the corresponding outputs, and the transition rules. By using a structured prompt template, *CoT prompting model* maintains the integrity of the original instruction while ensuring the symbolic logic is clearly communicated.

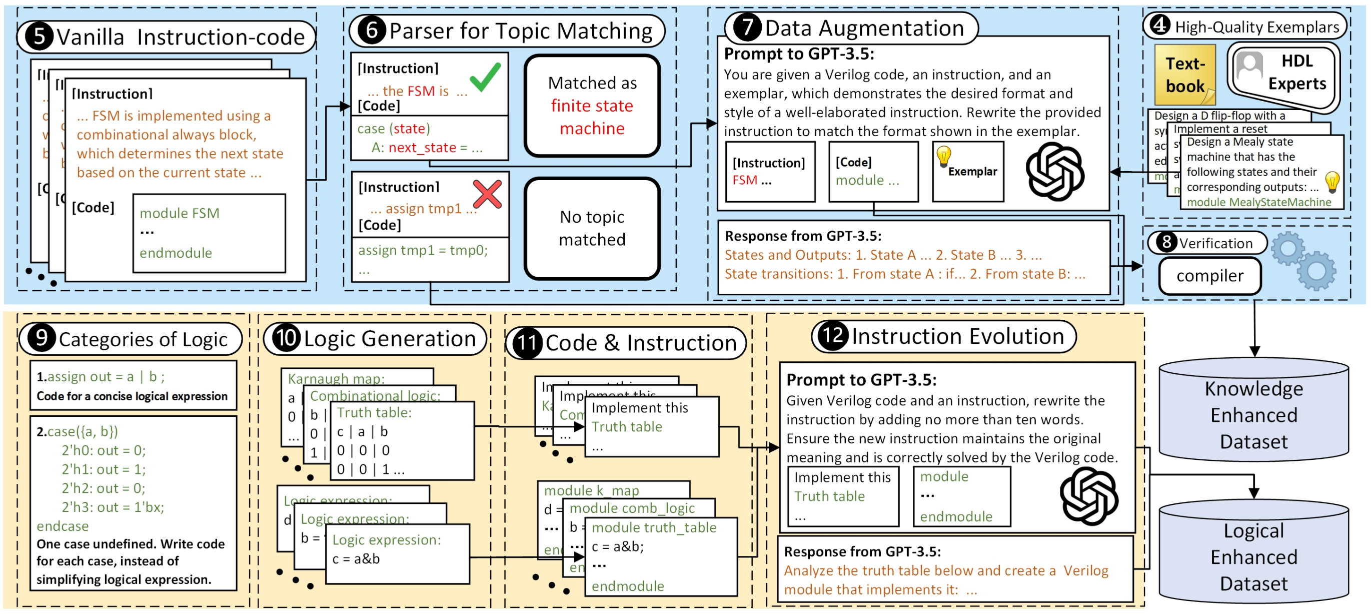
❸ **Add Module Header.** Finally, the interpreted instruction is checked to determine whether it contains a complete Verilog module header. If the module header is missing, *CoT prompting model* appends an appropriate module header to define the module name, inputs and outputs, standardizing the instruction.

Table III shows examples for SI-CoT. Incorporating symbolic interpretation within the CoT framework ensures a more reliable translation of various input modalities, reducing the risk of symbolic hallucination and improving the overall coherence between the input specifications and the generated code.

C. *Knowledge-Enhanced Dataset Generation*

As shown in the upper (blue shaded) part of Fig. 2, to mitigate knowledge hallucination, we develop a *knowledge enhanced dataset* generation process, which aligns the LLM-generated data with the practices of HDL engineers (human).

❹ **High-Quality Exemplars.** We curate high-quality exemplars that reflect digital design conventions and Verilog-specific attributes. These exemplars are derived from textbook exercises [19], [23]–[25] and manually designed examples that cover a wide range of Verilog knowledge. They include conventions for commonly implemented modules such as finite state machines (FSMs), clock dividers, counters, shift registers, and arithmetic logic units (ALUs). Additionally, they incorporate critical Verilog attributes like reset mechanisms (synchronous reset *vs.* asynchronous reset), clocking and edge sensitivity (positive edge *vs.* negative edge), and enable signals (active-high *vs.* active-low enable). These exemplars serve as the foundation for constructing HDL-aligned instruction-code pairs, ensuring that



5 Vanilla Instruction-code
[Instruction]
... FSM is implemented using a combinational always block, which determines the next state based on the current state ...
[Code]
module FSM
...
endmodule
6 Parser for Topic Matching
[Instruction]
... the FSM is ...
[Code]
case (state)
A: next_state = ...
Matched as finite state machine
[Instruction]
... assign tmp1 ...
[Code]
assign tmp1 = tmp0;
...
No topic matched
7 Data Augmentation
Prompt to GPT-3.5:
You are given a Verilog code, an instruction, and an exemplar, which demonstrates the desired format and style of a well-elaborated instruction. Rewrite the provided instruction to match the format shown in the exemplar.
[Instruction] FSM ...
[Code] module ...
Exemplar
Response from GPT-3.5:
States and Outputs: 1. State A ... 2. State B ... 3. ...
State transitions: 1. From state A : if... 2. From state B: ...
4 High-Quality Exemplars
Text-book
HDL Experts
Design a D flip-flop with a
Implement a reset
Design a Mealy state machine that has the following states and their corresponding outputs: ...
module MealyStateMachine
8 Verification
compiler
Knowledge Enhanced Dataset
Logical Enhanced Dataset
9 Categories of Logic
1.assign out = a | b ;
Code for a concise logical expression
2.case({a, b})
2'h0: out = 0;
2'h1: out = 1;
2'h2: out = 0;
2'h3: out = 1'bx;
endcase
One case undefined. Write code for each case, instead of simplifying logical expression.
10 Logic Generation
Karnaugh map:
Combinational logic:
Truth table:
c | a | b
0 | 0 | 0
0 | 0 | 1 ...
Logic expression:
Logic expression:
c = a&b
11 Code & Instruction
Implement this
Implement this Truth table
...
module k_map
module comb_logic
module truth_table
c = a&b;
...
endmodule
12 Instruction Evolution
Prompt to GPT-3.5:
Given Verilog code and an instruction, rewrite the instruction by adding no more than ten words. Ensure the new instruction maintains the original meaning and is correctly solved by the Verilog code.
Implement this Truth table ...
module ... endmodule
Response from GPT-3.5:
Analyze the truth table below and create a Verilog module that implements it: ...

Fig. 2: **Generation flow for *knowledge enhanced dataset* (K-dataset) and *logical enhanced dataset* (L-dataset)**. K-dataset and L-dataset are shuffled and combined as KL-dataset to fine-tune a CodeGen-LLM in HAVEN.

the generated instructions follow the questioning style of HDL engineers while embodying high-quality implementations.

❺ **Vanilla Instruction-code Pairs.** We collect approximately 550,000 Verilog code samples from public GitHub repositories. Following the methodology adopted in previous works [17], [22], we use GPT-3.5 to generate vanilla instructions for these code samples. By "vanilla instructions," we refer to basic, general-purpose instructions. These vanilla instruction-code pairs form a foundational dataset, which we refer to as the vanilla dataset. However, they often fall short of the precision and rigor demanded by HDL engineering standards.

❻ **Parser for Topic Matching.** The parser `slang` [26] is used to identify topics and attributes within the vanilla instruction-code pairs. These topics and attributes are matched with our curated exemplars to ensure each code sample aligns with specific Verilog conventions or attributes. If a vanilla pair lacks identifiable topics or attributes, it can still contribute to the dataset by mitigating the Verilog syntax misapplications.

❼ **Data Augmentation.** Given a related topic exemplar, GPT-3.5 rewrites the vanilla instruction to closely align it with the high-quality exemplar. If a vanilla instruction is associated with multiple exemplars, it is rewritten separately for each exemplar.

❽ **Verification.** To ensure the validity of the generated instruction-code pairs, we compile the Verilog code using an industry-standard Verilog compiler. This verification step ensures erroneous or incomplete pairs are filtered out.

D. Logical Enhanced Dataset Generation

As shown in the lower (yellow shaded) part of Fig. 2, we develop an L-dataset generation process to mitigate logical hallucination and enhance accurate logical reasoning.

❾ **Two Categories of Logical Reasoning in Verilog.** In Verilog, logical reasoning comes in twofold: finding the most concise logical expression, and faithfully implementing the logic when no concise expression is available. We generate datasets that cover both scenarios, allowing the fine-tuned model to select the appropriate approach during inference.

❿ **Generate Logical Expressions and Input-Output Values.** We begin by developing scripts that produce a wide range of logical expressions and their associated input-output mappings. The generated input-output values are assigned with a typical logic problem encountered in Verilog, *e.g.*, Karnaugh maps and combinational logic. These values serve as a basis for ensuring that the model understands the logical relationships in Verilog.

⓫ **Integration of code and instructions.** The next step is to incorporate the generated logical expressions and input-output values into pre-designed code templates and instruction templates. By embedding logical expressions into these templates, we create scenarios that reflect both straightforward and complex logical reasoning tasks.

⓬ **Instruction Evolution.** Instruction evolution [5] is frequently employed to generate diverse and complex instructions. To introduce linguistic variety, we use GPT-3.5 to rewrite the original instructions while ensuring the semantic core is retained. The modifications are constrained to adding or removing no more than ten words, preserving the logical structure. This step enhances the diversity of the dataset, improving the robustness of the model against overfitting.

By following the generation flow shown in Fig. 2, we obtain approximately 43k valid vanilla instruction-code pairs, 14k HDL-aligned instruction-code pairs for mitigating knowledge hallucination (*i.e.*, K-dataset), and 5k logically enhanced instruction-code pairs for mitigating logical hallucination (*i.e.*,

TABLE IV: **Comparison of HAVEN against various baseline models.**

	Model	Open source	Model Size	VerilogEval v1 (%)				RTLLM v1.1 (%)		VerilogEval v2 (%)	
				Machine		Human		Syntax pass@5	Func. pass@5	Task: Spec.-to-RTL	
										0-Shot (n=1)	
				pass@1	pass@5	pass@1	pass@5			pass@1	pass@5
General LLM	GPT-3.5	✗	n/a	46.7	69.1	26.7	45.8	89.7	37.9	n/a	n/a
	GPT-4	✗	n/a	60.0	70.6	43.5	55.8	100.0	65.5	44.2	n/a
	Starcoder [14]	✓	15B	46.8	54.5	18.1	26.1	93.1	27.6	n/a	n/a
	CodeLlama [27]	✓	7B	43.1	47.1	18.2	22.7	86.2	31.0	n/a	n/a
	DeepSeek-Coder [28]	✓	6.7B	52.2	55.4	30.2	33.9	93.1	44.8	28.2	n/a
	CodeQwen [29]	✓	7B	46.5	54.9	22.5	26.1	86.2	41.4	n/a	n/a
LLM for Verilog CodeGen	ChipNeMo [30]	✗	13B	43.4	n/a	22.4	n/a	n/a	n/a	n/a	n/a
	Thakur et al. [31]	✓	16B	44.0	52.6	30.3	43.9	86.2	24.1	n/a	n/a
	RTLCoder-Mistral [21]	✓	7B	62.5	72.2	36.7	45.5	96.6	48.3	n/a	n/a
	RTLCoder-DeepSeek [21]	✓	6.7B	61.2	76.5	41.6	50.1	93.1	48.3	36.5	n/a
	BetterV-CodeLlama [32]	✗	7B	64.2	75.4	40.9	50.0	n/a	n/a	n/a	n/a
	BetterV-DeepSeek [32]	✗	6.7B	67.8	79.1	45.9	53.3	n/a	n/a	n/a	n/a
	BetterV-CodeQwen [32]	✗	7B	68.1	79.4	46.1	53.7	n/a	n/a	n/a	n/a
	AutoVCoder-CodeLlama [22]	✗	7B	63.7	72.9	44.5	52.8	93.1	48.3	n/a	n/a
	AutoVCoder-DeepSeek [22]	✗	6.7B	69.0	79.3	46.9	53.7	100.0	51.7	n/a	n/a
	AutoVCoder-CodeQwen [22]	✗	7B	68.7	79.9	48.5	55.9	100.0	51.7	n/a	n/a
	OriGen-DeepSeek-7B-v1.5⋆ [17]	✓	7B	74.1	82.4	54.4	60.1	n/a	65.5	n/a	n/a
Ours	HAVEN-CodeLlama	✓	7B	74.7	80.0	51.3	59.0	95.4	54.7	46.4	55.8
	HAVEN-DeepSeek	✓	6.7B	78.8	84.5	57.3	64.2	92.8	66.0	58.3	63.4
	HAVEN-CodeQwen	✓	7B	77.3	81.2	61.1	64.8	92.8	62.2	54.6	62.9

†: ranking of performance as 🏆 1st, 🏆 2nd, 🏆 3rd place;
⋆: Only the base model of OriGen is DeepSeek-v1.5(7B), while the other works utilize DeepSeek-v1(6.7B)
Note: To the best of our knowledge, ChipNeMo [30], BetterV [32], and AutoVCoder [22] are not publicly accessible at the time of writing.

L-dataset). They are used to fine-tune the *CodeGen-LLM*.

IV. EXPERIMENTS

A. *Experiment setup*

Models and Platform. We utilize CodeLlama-7b-Instruct [27], Deepseek-Coder-6.7b-Instruct [28], and CodeQwen1.5-7B-Chat [29] as base models. One model is used for SI-CoT , fine-tuning and code generation. Fine-tuning is performed on two Nvidia A100-80GB GPUs for 3 epochs. We apply the AdamW optimizer with a cosine learning rate scheduler, initializing with 15 warm-up iterations with a learning rate of 5e-5. The global batch size is set to 256. Following RTLCoder [21], we set the temperature of each model to 0.2, 0.5 and 0.8, reporting the best performance.

Benchmark and Evaluation Metrics. Evaluations are performed on: VerilogEval v1 [10], RTLLM v1.1 [33], and the newly introduced VerilogEval v2 [15]. VerilogEval v1 is divided into two distinct components: VerilogEval-machine, which includes 143 Verilog generation tasks created using GPT-based models, and VerilogEval-human, a collection of 156 manually crafted problems. RTLLM v1.1 consists of 29 tasks focused on RTL design. VerilogEval v2 builds on the original VerilogEval-Human, extending its scope to encompass specification-to-RTL design tasks [15], whose prompt style is as a chat bot, with well-defined "Question" and "Answer".

We evaluate design performance from two key perspectives: *syntax correctness* and *functional correctness*. Both correctness are measured by the pass@k metric [8], which estimates the proportion of problems that can be solved in at least one of k attempts ($k \in \{1, 5\}$). It is expressed as:

$$\text{pass@}k := \mathbb{E}\left[1 - \binom{n-c}{k} \Big/ \binom{n}{k}\right] \quad (1)$$

where $n \geq k$ denotes the total number of trials per problem, and c represents the number of trials that passed the functional check. We set $n = 10$ in experiments.

B. *Comparison with Competing Works*

We evaluate the syntactic and functional correctness of HAVEN, which is open-source and no larger than 7B, on three benchmarks. Results are tabulated in Table IV, which indicate HAVEN performs best on three benchmarks in functional correctness. Specifically, HAVEN-DeepSeek performs 4.7% and 2.1% higher than OriGen in pass@1 and pass@5 respectively on VerilogEval-Machine. HAVEN-CodeQwen outperforms OriGen by 6.7% and 4.7% in pass@1 and pass@5 respectively on VerilogEval-Human, whose tasks are more aligned with HDL engineers and more closer to real-world practices. HAVEN-DeepSeek on RTLLM v1.1 performs the best in functional pass@5, which outperforms both OriGen and GPT-4 by 0.5%. Although AutoVCoder [22] is fine-tuned on a larger dataset based on around one million Verilog modules and 50,000 synthetic code samples, our HAVEN-DeepSeek, fine-tuned on just 62,000 data samples, achieves only 4-6% lower in terms of syntax pass@5. Our models also perform well on VerilogEval v2, whose prompts are more conform to expectations of HDL engineers. HAVEN-DeepSeek achieves 58.3% and 63.4% in pass@1 and pass@5 respectively. Additionally, after fine-tuning, CodeLlama performs worse than the other two models, which aligns with the experimental results reported in other study [22]. These results demonstrate that HAVEN exhibits significant superiority in Verilog code generation and meets practical needs of HDL engineers.

C. *Evaluation on Symbolic Modalities*

To evaluate how HAVEN handles symbolic modalities, we curate a set of 44 tasks from VerilogEval(v1)-Human, which

TABLE V: **Evaluation on Symbolic Modalities**. P/T denotes pass cases/total cases. PR is the pass rate.

Model	Truth Table	Waveform	State Diagram	Overall
	P/T (PR)	P/T (PR)	P/T (PR)	Pass@1
RTLCoder [21]	1/10(10.0%)	3/13(23.1%)	3/21 (14.3%)	15.9%
OriGen [17]	2/10(20.0%)	3/13(23.1%)	5/21(23.8%)	22.7%
GPT-4	2/10(20.0%)	3/13(23.1%)	5/21(23.8%)	22.7%
DeepSeek-Coder-V2	3/10(30.0%)	3/13(23.1%)	9/21(42.9%)	34.1%
HAVEN-CodeQwen	6/10(60.0%)	4/13(30.8%)	11/21(52.4%)	47.4%

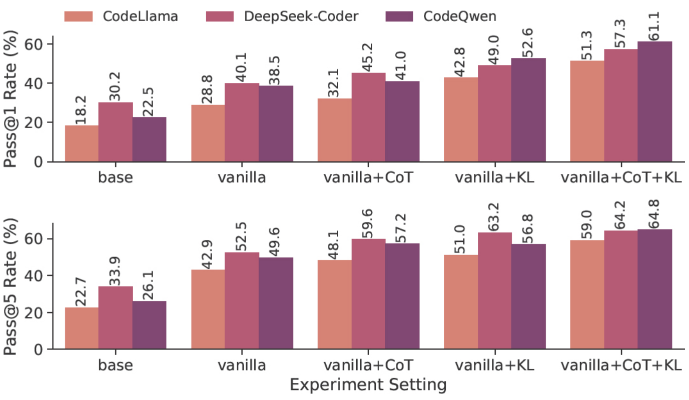


Fig. 3: **Ablation Study of techniques adopted in HAVEN.** Pass@1/5 are reported on VerilogEval(v1)-Human dataset.

span three categories that frequently induce hallucinations: *truth tables(10)*, *waveform charts(13)*, and *state diagrams(21)*.

In this experiment, we compare our model, HaVen-CodeQwen, against commercial LLMs (GPT-4 and DeepSeek-Coder-V2 [34]) and open-source Verilog CodeGen models (RTLCoder [21] and OriGen [17]), on the set of 44 tasks.

The results, shown in Table V, reveal an interesting trend: GPT-4 and OriGen exhibit similar performance, while DeepSeek-Coder-V2 outperforms both in all three categories, though it remains second to our model. By breaking down performance in each modality, HAVEN shows its superior performance in handling these symbolic modalities, a common pain point for LLMs. HAVEN-CodeQwen outperforms all models in three categories, achieving the highest overall pass@1 of 47.7%. This underscores the effectiveness of HAVEN in generating accurate Verilog code from complex instructions.

D. Ablation study

Effectiveness of Techniques is evaluated on the VerilogEval(v1)-Human (Fig. 3), with five settings:

- ☐ Base: Original pre-trained LLMs with no modifications;
- ☐ Vanilla: Base LLMs fine-tuned only by vanilla dataset (Section III-C).
- ☐ Vanilla+CoT: Base LLMs fine-tuned by vanilla dataset, then prompted by SI-CoT instructions.
- ☐ Vanilla+KL: CodeGen LLMs fine-tuned by KL-dataset.
- ☐ Vanilla+CoT+KL: CodeGen LLMs fine-tuned by KL-dataset, then prompted by SI-CoT instructions.

The incorporation of the proposed SI-CoT and KL-dataset in HAVEN shows effectiveness across three base LLMs. Solely adopting SI-CoT improve the Pass@1 and Pass@5 by 3.6% and 6.6% on average, while fine-tuning LLMs with KL-dataset improves these metrics by 12.3 % and 8.7% on average. When combining SI-CoT and KL-dataset together, we still observe an improvement, which demonstrates that they supplement each other.

TABLE VI: **Evaluation of SI-CoT on commercial LLMs.**

	GPT-4o mini	GPT-4	DeepSeek-Coder-V2
Pass@1 (w SI-CoT)	22.7%	22.7%	34.1%
Pass@1 (w/o SI-CoT)	31.8%	34.1%	45.5%

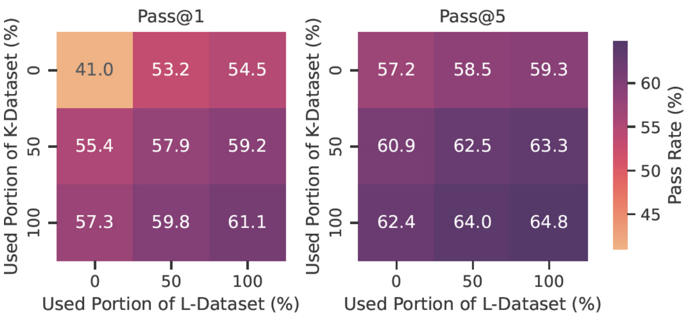


Fig. 4: **Ablation Study of composition of KL-dataset**.

Analysis of SI-CoT is further extended to commercial LLMs to assess the effectiveness of SI-CoT. We select a set of 44 tasks from VerilogEval(v1)-Human, same as we conducted in Table V. For fair comparison, all LLMs are prompted with the same SI-CoT instructions produced by the base model Code-Qwen [29]. The results in Table VI shows SI-CoT directly helps with CodeGen LLM even without fine-tuning. Moreover, the results reveal an intriguing finding: GPT-4o mini, despite being cheaper than GPT-4, delivers comparable performance. Meanwhile, DeepSeek-Coder-V2, the most affordable of the three commercial LLMs, demonstrates impressive results, matching GPT-4's performance with SI-CoT, even without using SI-CoT.

Analysis of KL-dataset. As depicted in Fig. 4, we use {0%, 50%, 100%} portions from K-dataset and L-dataset and mix them to fine-tune the *CodeGen-LLM* (*i.e.*, CodeQwen). Such experiment is to evaluate the effectiveness of K-dataset and L-dataset, using VerilogEval(v1)-Human. The results reveal that both K-dataset and L-dataset contribute to the pass@k improvement. The improvement resulting from the K-dataset is higher, where the K-dataset contains more data than L-dataset could be the reason. In addition, further enlarging the samples in KL-dataset can still be beneficial to optimize HAVEN.

V. CONCLUSION

In this work, we present HAVEN, a novel LLM framework for Verilog code generation. We introduce a hallucination taxonomy, including symbolic, knowledge and logical hallucinations. HAVEN addresses these challenges through a three-stage methodology focused on reducing hallucinations in LLM-generated code while enhancing adherence to the practices of HDL engineers. Experimental results demonstrate that HAVEN outperforms existing state-of-the-art methods and that our three techniques are effective individually.

References

[1] Z. Luo, C. Xu, P. Zhao, Q. Sun, X. Geng, W. Hu, C. Tao, J. Ma, Q. Lin, and D. Jiang, "Wizardcoder: Empowering code large language models with evol-instruct," 2024. [Online]. Available: https://iclr.cc/virtual/2024/poster/18519

[2] N. Muennighoff, Q. Liu, A. Zebaze, Q. Zheng, B. Hui, T. Y. Zhuo, S. Singh, X. Tang, L. von Werra, and S. Longpre, "Octopack: Instruction tuning code large language models," 2024. [Online]. Available: https://iclr.cc/virtual/2024/poster/17875

[3] OpenAI *et al.*, "Gpt-4 technical report," OpenAI, Tech. Rep., 2024.

[4] Y. Wei, Z. Wang, J. Liu, Y. Ding, and L. Zhang, "Magicoder: Source code is all you need," *ICML*, 2024. [Online]. Available: https://api.semanticscholar.org/CorpusID:265609970

[5] Q. S. Can Xu *et al.*, "WizardLM: Empowering large language models to follow complex instructions," in *International Conference on Learning Representations (ICLR)*, 2024. [Online]. Available: https://iclr.cc/virtual/2024/poster/19164

[6] T. Zheng, G. Zhang, T. Shen, X. Liu, B. Y. Lin, J. Fu, W. Chen, and X. Yue, "Opencodeinterpreter: Integrating code generation with execution and refinement," 2024. [Online]. Available: https://aclanthology.org/2024.findings-acl.762.pdf

[7] GitHub, "Github copilot," https://copilot.github.com, 2021.

[8] M. Chen, J. Tworek, H. Jun, Q. Yuan, and et al., "Evaluating large language models trained on code," 2021. [Online]. Available: https://arxiv.org/abs/2107.03374

[9] J. Liu, C. S. Xia, Y. Wang, and L. Zhang, "Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation," in *Proceedings of the 37th International Conference on Neural Information Processing Systems*, ser. NIPS '23. Red Hook, NY, USA: Curran Associates Inc., 2024.

[10] M. Liu, N. Pinckney, B. Khailany, and H. Ren, "Verilogeval: Evaluating large language models for verilog code generation," in *2023 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2023. [Online]. Available: https://api.semanticscholar.org/CorpusID:261822682

[11] Z. Ji, N. Lee, R. Frieske, T. Yu, D. Su, Y. Xu, E. Ishii, Y. J. Bang, A. Madotto, and P. Fung, "Survey of hallucination in natural language generation," *ACM Comput. Surv.*, vol. 55, no. 12, mar 2023. [Online]. Available: https://doi.org/10.1145/3571730

[12] Y. Tian, W. Yan, Q. Yang, X. Zhao, Q. Chen, W. Wang, Z. Luo, L. Ma, and D. Song, "Codehalu: Investigating code hallucinations in llms via execution-based verification," 2024. [Online]. Available: https://arxiv.org/abs/2405.00253

[13] F. Liu, Y. Liu, L. Shi, H. Huang, R. Wang, Z. Yang, L. Zhang, Z. Li, and Y. Ma, "Exploring and evaluating hallucinations in llm-powered code generation," 2024. [Online]. Available: https://arxiv.org/abs/2404.00971

[14] R. Li *et al.*, "Starcoder: may the source be with you!" *arXiv preprint*, 2023. [Online]. Available: https://arxiv.org/abs/2305.06161

[15] N. Pinckney, C. Batten, M. Liu, H. Ren, and B. Khailany, "Revisiting verilogeval: Newer llms, in-context learning, and specification-to-rtl tasks," *arXiv preprint arXiv:2408.11053*, 2024. [Online]. Available: https://doi.org/10.48550/arXiv.2408.11053

[16] Y. Tsai, M. Liu, and H. Ren, "Rtlfixer: Automatically fixing rtl syntax errors with large language model," New York, NY, USA, 2024. [Online]. Available: https://doi.org/10.1145/3649329.3657353

[17] F. Cui, C. Yin, K. Zhou, Y. Xiao, G. Sun, Q. Xu, Q. Guo, D. Song, D. Lin, X. Zhang *et al.*, "Origen: Enhancing rtl code generation with code-to-code augmentation and self-reflection," *International Conference on Computer-Aided Design*, 2024. [Online]. Available: https://api.semanticscholar.org/CorpusID:271334210

[18] N. A. Badiger, J. Muragod, P. Pattar, P. Gundlur, and S. Bhadri, "Fpga implementation of image enhancement using verilog hdl," *International Research Journal of Engineering and Technology (IRJET)*, 2020. [Online]. Available: https://www.irjet.net/archives/V7/i6/IRJET-V7I61064.pdf

[19] M.-B. Lin, *Digital System Designs and Practices: Using Verilog HDL and FPGAs*. Wiley Publishing, 2008. [Online]. Available: https://dl.acm.org/doi/10.5555/1502316

[20] https://github.com/.

[21] S. Liu, W. Fang, Y. Lu, Q. Zhang, H. Zhang, and Z. Xie, "Rtlcoder: Outperforming GPT-3.5 in design RTL generation with our open-source dataset and lightweight solution," in *IEEE International Workshop on LLM-Aided Design (LAD)*, 2024. [Online]. Available: https://api.semanticscholar.org/CorpusID:266209884

[22] 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 *Proceedings of the International Conference on Computer Design (ICCD)*, 2024. [Online]. Available: https://api.semanticscholar.org/CorpusID:271516210

[23] M. D. Ciletti, *Advanced Digital Design with the Verilog HDL*, 2nd ed. USA: Prentice Hall Press, 2010.

[24] S. Palnitkar, *Verilog® hdl: a guide to digital design and synthesis, second edition*, 2nd ed. USA: Prentice Hall Press, 2003.

[25] B. J. LaMeres, *Quick Start Guide to Verilog*, 1st ed. Springer Publishing Company, Incorporated, 2019.

[26] M. Popoloski, "Slang," 2023, online. Available: https://github.com/MikePopoloski/slang.

[27] B. Roziere *et al.*, "Code llama: Open foundation models for code," *arXiv preprint*, 2023. [Online]. Available: https://arxiv.org/abs/2308.12950

[28] D. Guo *et al.*, "Deepseek-coder: When the large language model meets programming – the rise of code intelligence," 2024. [Online]. Available: https://arxiv.org/abs/2401.14196

[29] J. Bai *et al.*, "Qwen technical report," 2023. [Online]. Available: https://arxiv.org/abs/2309.16609

[30] M. Liu *et al.*, "Chipnemo: Domain-adapted llms for chip design," *arXiv preprint arXiv:2311.00176*, 2023. [Online]. Available: https://arxiv.org/abs/2311.00176

[31] 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 Transactions on Design Automation of Electronic Systems*, vol. 29, no. 3, pp. 1–31, 2024. [Online]. Available: https://doi.org/10.1145/3643681

[32] Z. Pei, H. Zhen, M. Yuan, Y. Huang, and B. Yu, "Betterv: Controlled verilog generation with discriminative guidance," in *International Conference on Machine Learning (ICML)*, Vienna, July 2024, pp. 21–27. [Online]. Available: https://api.semanticscholar.org/CorpusID:267500201

[33] Y. Lu, S. Liu, Q. Zhang, and Z. Xie, "Rtllm: An open-source benchmark for design rtl generation with large language model," in *Proceedings of the Asia and South Pacific Design Automation Conference (ASP-DAC)*, 2024. [Online]. Available: https://doi.org/10.1109/ASP-DAC58780.2024.10473904

[34] DeepSeek-AI, Q. Zhu, D. Guo, Z. Shao, and et al., "Deepseek-coder-v2: Breaking the barrier of closed-source models in code intelligence," 2024. [Online]. Available: https://arxiv.org/abs/2406.11931